1-1-2016

Technical Debt Decision-Making Framework

Zadia Codabux

Technical debt decision-making framework

By

Zadia Codabux

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2016

Technical debt decision-making framework

By

Zadia Codabux

Approved:

Byron J. Williams
(Major Professor)

Gary L. Bradshaw
(Committee Member)

J. Edward Swan II
(Committee Member)

Murray Cantor
(Committee Member)

T. J. Jankun-Kelly
(Graduate Coordinator)

Jason M. Keith
Dean
Bagley College of Engineering

Name: Zadia Codabux

Date of Degree: December 9, 2016

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Byron J. Williams

Title of Study: Technical debt decision-making framework

Pages of Study: 155

Candidate for Degree of Doctor of Philosophy

Software development companies strive to produce high-quality software. In commercial software development environments, due to resource and time constraints, software is often developed hastily which gives rise to technical debt. Technical debt refers to the consequences of taking shortcuts when developing software. These consequences include making the system difficult to maintain and defect prone. Technical debt can have financial consequences and impede feature enhancements. Identifying technical debt and deciding which debt to address is challenging given resource constraints. Project managers must decide which debt has the highest priority and is most critical to the project. This decision-making process is not standardized and sometimes differs from project to project. My research goal is to develop a framework that project managers can use in their decision-making process to prioritize technical debt based on its potential impact. To achieve this goal, we survey software practitioners, conduct literature reviews, and mine

software repositories for historical data to build a framework to model the technical debt decision-making process and inform practitioners of the most critical debt items.

Key words: technical debt, decision making, prioritization, predictive analytics, bayesian networks, interviews, questionnaires

DEDICATION

To Mum and Dad.

ACKNOWLEDGEMENTS

First and foremost, I thank God who made me who I am today.

This dissertation could not have been accomplished without the support and encouragement of many people. I would like to acknowledge those people who have helped me during this magnificent journey.

I would especially like to thank my advisor, Byron Williams, for his continuous support, encouragement, tremendous patience, and valuable suggestions that enabled me to accomplish this endeavor. I am grateful for having an exceptional doctoral committee. I would like to thank Dr. Bradshaw for his generous time and unfailing assistance. I greatly appreciate that Dr. Cantor, my IBM mentor and committee member who believed that I could accomplish great things at a time when I had no idea what I was doing. As you recently mentioned, I have come a long way since we first talked. You have always encouraged me to strive, pricked my curiosity and viewed my work from an industrial perspective. I thank Dr. Swan for tickling my imagination and making me think outside the box. I am the researcher I am today because of you all.

I am grateful to Dr. Reese who was there for me every step of the way. I wholeheartedly appreciate everything she has done for me during my doctoral program. I would like to express my heartfelt thanks to Dr. Ed Allen for giving me this opportunity. Since the day

I stepped foot in the CSE department, Dr. Allen has always been patient and his door has been opened for any advice.

This dissertation would not have been possible without the generous funding of the Fulbright Foreign Student Program Fellowship and the IBM Ph.D. Fellowship. The CSE department at MSU and the Bridge funding from the Bagley College of Engineering were kind enough to support me financially during my last year at MSU. I am grateful to CRA-W, the Anita Borg Institute, and NSF for the numerous travel awards during my doctoral program.

This dissertation benefitted from the strong collaboration with industry practitioners and volunteer participants in my studies. Therefore, I would like to extend my sincere thanks to all my collaborators and the technical debt research community. My appreciation goes to the ESE research group for their suggestions and comments during our weekly research meetings. My appreciation goes to my lab mates as the days in the lab would have seemed longer without you guys. Thank you for your support and the fruitful discussions throughout the years. In addition, I am much obliged to the Bagley College of Engineering writing tutors for their patience in reviewing this dissertation and many of my publications throughout the years.

I am deeply indebted to my family members who never stopped believing in me, even at times when I had no faith in myself. I am eternally grateful to my father who allowed me to pursue my dream. I would like to express my heartfelt thanks to my mother, my role model and one of the smartest and most sensible people that I know, for her unfailing good advice and not for a single day making me feel that we were on opposite sides of

the world. Words are powerless to express my gratitude to my husband without whom I could not have completed this dissertation. I thank him for his endless encouragement, confidence, motivation, support and for always being there for me.

All my love and thanks to my sister and brother who have always given me the psychological strength and moral support from afar. My nieces and nephews have been a bright light during my doctoral program. Their innocent smiles and admiration have often been a source of motivation for me to work harder and make them proud of me.

The hardest part of studying abroad was being away from my family, but thanks to all the wonderful people of Starkville, I felt home.

Last but not least, I would like to thank my well-wishers. I know that you have always been praying hard for my success.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Software is pervasive and has a critical role in many industries ranging from aerospace, military, medical, transportation, financial and others. Therefore, it is of the utmost importance that software that is developed is of high quality. In reality, software development is prone to failure. Over time, software suffers from a steady degradation of quality, and it becomes increasingly difficult and costly to maintain the software. In fact, a study by Cambridge University reported that software bugs cost the economy $312 billion annually.[1] These bugs cost the US economy about $59.5 billion annually, representing about 0.6% of the gross domestic product (GDP)[2].

Fixing severe problems after delivery can be 100 times more expensive than fixing pre-delivery while non-severe defects can be twice as expensive [77]. Furthermore, most software development costs occur after the product is released. In addition to being costly, corrective maintenance work can be a blow to the reputation of the organization. If the defects are trivial, the customer might think the software development company has been negligent and not tested the software thoroughly. If the defects are serious and they affect

[1] Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year, http://markets.financialcontent.com/stocks/news/read/23147130/Cambridge_University_Study_States_Software_Bugs_Cost_Economy$312_Billion_Per_Year (accessed April 16, 2013)

[2] The Economic Impacts of Inadequate Infrastructure for Software Testing, https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf (accessed April 16, 2013)

critical functions, the customer might go as far as pursuing legal action against the software development company, causing serious damage to its reputation.

One of the root causes of this failure is the use of sequential design processes for building complex software-intensive systems. Sequential processes work when the requirements are defined upfront and when the remaining software development activities are instituted based on the initial requirements (e.g. design, implementation, testing). Traditional software development processes, such as the waterfall model, are not the most appropriate when business needs and technology change rapidly. Many development groups often deviate from this normative waterfall process. Their focus shifts to the product and the customer needs rather than the plan. This shift gives rise to agile software development [75].

One of the primary benefits of agile software development is the quick release of software functionality. However, the focus on functionality often lessens the focus on design, good programming practice, test coverage, etc. By focusing on functionality, the developer is then obliged to go back and complete these items neglected for the sake of functionality. This phenomenon is known as technical debt.

While the concept of technical debt has been existent for some time, it has been the adoption of agile development methods that has given the term its visibility. Agile methods started to grow in popularity in 2001 following the signing of the Agile Manifesto. Since then, technical debt has become an increasingly important concept in the software engineering research community.

The technical debt concept was coined in the 1992 OOPSLA report by Ward Cunningham and is one of the most widely accepted descriptions of technical debt [20]:

"Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt."

However, technical debt is no longer restricted to comparisons with source code. In fact, technical debt is used to describe situations when developers accept sacrifices in one dimension of development (e.g. software quality) in order to optimize another dimension (e.g. implementing necessary features before a deadline) [91]. Its application has evolved and has been extended to other artifacts (e.g. design, testing, and documentation) of the software development lifecycle.[3]

The definition provided by Steve McConnell at the Managing Technical Debt workshop is more in line with the evolution of technical debt since its conception. We used McConnell's definition as the baseline definition of technical debt in this study.

"[Technical Debt is] a design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)." [4]

As participants of the 16162 Dagstuhl Seminar on Managing Technical Debt[5], we helped refine the definition of technical debt to restrict the scope of the metaphor to internal system qualities such as maintainability and evolvability.

[3] IEEE/Lockheed Martin Webinar on Identifying and Managing Technical Debt, http://www.nicozazworka.com/research/technical-debt/ (accessed Jun. 28, 2013)

[4] S. McConnell, "Managing technical debt," Fourth International Workshop on Managing Technical Debt, 2013, http://www.sei.cmu.edu/community/td2013/program/upload/TechnicalDebt-ICSE.pdf (accessed Jun. 28, 2013)

[5] Managing Technical Debt in Software Engineering, http://www.dagstuhl.de/no_cache/en/program/calendar/partlist/?semnr=16162&SUOG= (accessed Jul. 20, 2016)

"In software-intensive systems, technical debt is a design or implementation construct that is expedient in the short term but sets up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability, and evolvability."[6]

Similar to financial debt, technical debt needs to be "paid back" because otherwise, the interest (software maintenance) will increase and hinder new development. Technical debt is often the result of developers taking shortcuts (e.g. copy-paste code) to meet a deadline or ignorance (e.g. choosing the wrong design pattern). The impact of accumulated technical debt can be decreased productivity, increased cost, and, eventually, system degradation as elaborated in Section 3.4.

The bottom line is that technical debt needs to be addressed. This is often a long-term investment that is broken down into small chunks where a certain amount of technical debt items are tackled with an allocated time-frame (e.g. two days are dedicated to technical debt management per every two weeks). Therefore, it is important that managers optimize this allocated time to handle the most critical debt items.

The research goal of this dissertation is to develop a framework that project managers can use in their decision-making process to prioritize technical debt items.

To achieve that goal, we have devised the following research questions:

RQ1: What are the prevalent notions of technical debt?

RQ2: What are the different types of technical debt indicators?

RQ3: How do we build a framework to determine the most critical technical debt items?

[6]The 16162 definition, https://mtd2016dagstuhl.org/ (accessed Jul. 20, 2016)

RQ4: How effective is the framework?

This dissertation is two-fold. First, we conducted a literature review and empirical studies with software practitioners to understand the state of the art of technical debt in industry. Second, we use the knowledge acquired during our case studies coupled with machine learning and decision models in order to build a decision-making framework for technical debt.

The contributions of this dissertation can be summarized as follows:

- A comprehensive and customizable framework to manage technical debt items based on business objectives while assessing risk
- A prediction model that can be used independently of the entire framework (e.g. focus quality assurance effort)
- A technical debt taxonomy tree which classifies the types of technical debt
- A technical debt management tree which proposes criteria that project managers can use in their decision models when reasoning about technical debt

The rest of this dissertation is organized as follows. Chapter 2 highlights background and related work. Chapter 3 focuses on the literature review on technical debt organized in different categories. Chapter 4 presents the methodology including research goal, questions, and study design. Chapter 5 provides insights on the results. Chapter 6 elaborates on the results and lists the threats to validity. Finally, Chapter 7 concludes this dissertation and discusses future work.

CHAPTER 2

BACKGROUND

In this chapter, we describe some important concepts related to this research. First, we elaborate on agile development methodology, which gives technical debt more visibility. Second, we describe some important metrics (traditional and object-oriented) defined in the literature that are relevant for this work.

2.1 Agile Practices and Technical Debt

In February 2001, seventeen practitioners met in Utah to discuss their work habits, the lightweight methodologies. This gave rise to agile software development, a reaction to traditional, plan-based software development. The practitioners wrote the "Manifesto for Agile Software Development"[1] which describes the four comparative values underlying the agile position:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

[1] Manifesto for Agile Software Development, http://www.agilemanifesto.org/(accessed March 5, 2012)

That is, while there is value in the items on the right, we value the items on the left more.

Agile development involves a set of practices that promotes an incremental delivery of software. Agile teams are small, self-organizing and cross-functional. Agile encourages rapid development and delivery while being flexible to changes. Williams and Cockburn state that agile development is about feedback and change and are developed to embrace, rather than reject, higher rates of change [87].

Agile software development has a huge impact on the way software is developed worldwide. Recently, agile processes have gained increasing adoption levels as shown in Figure 2.1[2] and have rapidly joined the mainstream of development approach. Companies that adopted agile reported that the main benefits obtained from implementing agile include the ability to manage change priorities, improved project visibility, and increased productivity [75].

There are many agile development methods (e.g. Scrum, Extreme Programming, Lean and Dynamic System Development Method amongst others). Most methods promote teamwork, customer involvement, and process adaptability throughout the life-cycle of the project. Methods for agile consist of a set of practices for software development that have been created by experienced practitioners.

In order to better understand agile in an industry context, we conducted a case study with an industrial partner during their implementation of agile development practices for a large software development division within the company [16].

[2]HP, "Agile is the new normal," https://www.hpe.com/h20195/v2/getpdf.aspx/4AA5-7619ENW.pdf?ver=1.0 (accessed September 10, 2016)

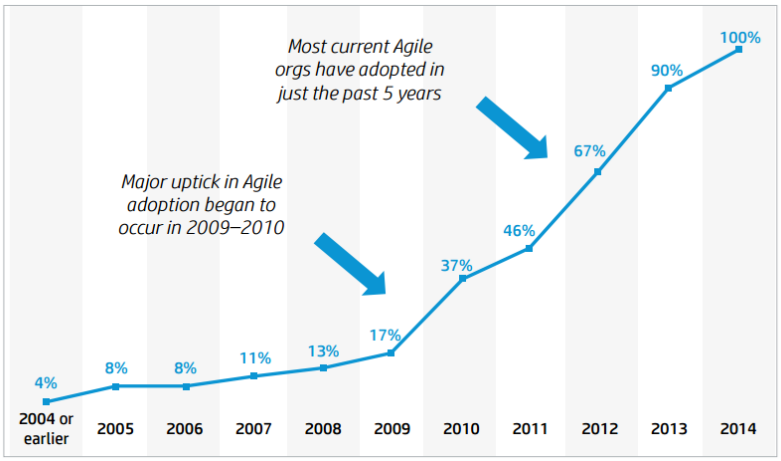

Figure 2.1

Agile Adoption over Time

Despite their recent agile adoption, the company is conscious of the impact of technical debts in the software development process, and therefore, some teams are dedicated to the reduction of debt.

This study increased our awareness of the importance of managing technical debt. Subsequently, we devised a method for addressing technical debt using a Quality Assurance (QA) classification scheme and focus on prevention, reduction, and containment activities [18].

Technical debt prevention activities aim to proactively reduce the chance of technical debts being injected in the software. Different strategies are proposed to handle the different sources of debt injections including pair programming, refactoring, and test-driven development.

One of the most common technical debt prevention techniques is refactoring. Refactoring is also commonly used as a technical debt reduction technique. Refactoring is a technique for restructuring code by improving its internal structure without affecting its external behavior. Refactoring improves not only aspects of code quality, but also productivity [62]. Pros of refactoring include improved code readability and reduced complexity. Refactoring thus makes the maintainability and extensibility of code easier. Examples of refactoring involve deleting unused code and duplicated code.

The aim of technical debt reduction activities is to remove as much of the injected debt as possible. There are numerous good practices in agile software development that can benefit technical debt reduction. Such agile practices include code reviews, static analysis tools, test automation, and frequent customer feedback.

Technical debts that bypass the debt prevention and reduction activities intentionally or unintentionally remain present in the software. It is a common practice to have working software deployed at a customer's site with known technical debt present. For these situations, technical debt containment is the quality assurance approach that must be taken. It is imperative to isolate the impact of known debts so that other parts of the software are not impacted. While there is a lack of techniques proposed in existing literature to contain technical debt, N-version programming (NVP) is one applicable approach.

2.2 Software Metrics

Metrics have traditionally been used as an indicator of software quality in the industry. In addition, metrics are widely used as a proxy for technical debt or tools measuring techni-

cal debt uses metrics in the calculation of their technical debt indicator [64, 58, 22, 7, 54]. In this section, we will present a brief overview of the different common metrics in the literature that have been used in the context of software quality over the years.

2.2.1 Traditional Metrics

Traditional software metrics that have been widely used among practitioners are lines of code, Halstead metrics (Software Science), and cyclomatic complexity. We defined these common traditional metrics as in Table 2.1.

Table 2.1

Traditional Metrics

Metric	**Description**
Line of Code (LOC)	Measure the size of a computer program by counting the number of lines in the source code
McCabe Cyclomatic Complexity	Indicate the complexity of a program based on graph theory [59]
Halstead Metrics	Measure a program module's complexity directly from source code, with emphasis on computational complexity [35]

2.2.2 Object Oriented Metrics

Object-Oriented metrics are increasingly being used to evaluate the quality of software. When code is analyzed for object-oriented metrics, the Chidamber-Kemerer (CK) suite [14] is a very common choice. We present the CK Metrics suite in Table 2.2.

Some of the metrics defined in this section will be used in the building and evaluation of the technical debt decision-making framework.

Table 2.2

Chidamber and Kemerer OO Metrics Suite

Metric	**Description**
Weighted Methods Per Class (WMC)	Measures the complexity of an individual class
Depth of Inheritance Tree (DIT)	Indicates the maximum depth of the inheritance graph of each class
Number of Children (NOC)	Measures the number of direct descendants for each class
Coupling between object classes (CBO)	Measures the number of classes to which a given class is coupled
Response For a Class (RFC)	Measures the number of methods that can potentially be executed in response to a message received by an object of that class
Lack of Cohesion in Methods (LCOM)	Measures the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables

CHAPTER 3

LITERATURE REVIEW

Since the first Managing Technical Debt workshop in 2010, research on technical debt has become more widespread. There has been a substantial increase in the number of studies conducted on the topic. Unlike traditional systematic literature reviews [50] [49], due to the lack of academic research in this area, we studied all the publications that were available on technical debt from 2010 to 2014. This chapter groups and summarizes some of the studies carried out in the field of technical debt. We start by describing technical debt taxonomies, followed by technical debt measurement and decision-making. Then we enumerate the impact of technical debt and describe empirical studies carried out with software practitioners.

3.1 Taxonomies

Several notable taxonomies have been developed for classifying technical debt.

McConnell classified technical debt as intentional and unintentional debt.[1] He defines unintentional debt as debt acquired unintentionally due to low quality work (e.g. an inexperienced programmer writing bad code). He describes intentional debt as debt acquired knowingly (e.g. a conscious decision to achieve some short-term objectives). Intentional

[1] S. McConnell, "Technical Debt," http://www.construx.com/10x_Software_Development/Technical_Debt/ (accessed Apr. 14, 2013)

debt is further broken down as short-term debt and long-term debt. Short-term debt is debt taken on tactically and reactively (e.g. as a last resort to get a release out of the door). Long-term debt is taken on strategically and proactively, especially in cases where the cost as at today is seen as expensive as the cost in the future.

Fowler[2] defines technical debt as either reckless or prudent and deliberate or inadvertent. The four quadrant's categorization of technical debt are as follows:

- Reckless and deliberate - debt created due to poor management where shortcuts are taken by the team instead of the right solution because they have to meet a deadline
- Prudent and deliberate - debt taken on after carefully analyzing the costs and consequences of shipping now
- Reckless and inadvertent - debt accumulated by developers who are incompetent and ignorant of coding practices
- Prudent and inadvertent - debt that refers to improvements that the team can make with experience and relevant knowledge

Rothman's classification[3] is analogous to life cycle phases: design, development and testing debt. Design debt refers to design which is not robust in certain areas. Development debt is described as missing code, and testing debt refers to tests not developed or run against the code.

[2]M. Fowler, "TechnicalDebtQuadrant," http://martinfowler.com/bliki/TechnicalDebtQuadrant.html (accessed Jan. 14, 2014)

[3]J. Rothman, "An Incremental Technique to Pay Off Testing Technical Debt," http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=11011 (accessed Apr. 14, 2013)

Alves et al. [2] proposed an ontology regrouping thirteen technical debt types, namely architecture debt, build debt, code debt, defect debt, design debt, documentation debt, infrastructure debt, people debt, process debt, requirement debt, service debt, test automation debt and test debt. We define these types of technical debt in Table 3.1.

To further advance research in technical debt, it is important to understand and distinguish between the different types of debt in software development. This distinction of the types of technical debt will contribute towards refining a definition for technical debt. However, technical debt taxonomy is an area of research in technical debt which is among the least studied. Many of the taxonomies presented were reported in well-respected software practitioner blogs. Therefore, we need more studies on the taxonomy of technical debt to help the community recognize the various types of debt and to pave the way toward a cohesive definition for technical debt.

3.2 Measurement

The baseline for measuring technical debt is the concept of principal and interest. Principal refers to the cost of completing the task at the present time. Interest is the extra cost added to the principal that will be needed to complete the task at a later time [76]. Cunningham introduced the concept of interest probability as the probability that the debt will make other tasks more expensive over time if it is not paid [20]. However, many more elaborate methods have been developed to quantify technical debt as discussed below.

Nugroho et al. [65] defined technical debt and interest as the cost to improve quality to an ideal level and additional cost of software maintenance for not achieving ideal quality,

Table 3.1

Types of Technical Debt

Types of Technical Debt	**Definition**
Architecture debt	Architecture debt refers to problems related to the project architecture e.g.,modularity violations
Build debt	Build debts refer to build related issues that complicate the build process
Code debt	Code debts refer to problems in the source code related to bad coding practices
Defect debt	Defect debts are known defects that have been postponed to be fixed later
Design debt	Design debt refers to debt accumulated due to the violations of good object-oriented design principles
Documentation debt	Documentation debts are missing, incomplete or inadequate documentation for software
Infrastructure debt	Infrastructure debt includes infrastructure issues that can potentially hinder development activities
People debt	People debts are people issues that can delay development activities
Process debt	Process debts refer to inefficient processes
Requirement debt	Requirement debts are tradeoffs made with respect to which requirements need to be implemented
Service debt	Service debts refer to the need for web service substitution, which can potentially introduce debt
Test automation debt	Test automation debts relate to automating the tests of previously developed functionality
Test debt	Test debts are tests that are planned but not run or other known shortcomings in the test suite

respectively. Technical debt was quantified as Repair Effort (RE) in terms of Rework Fraction (RF), Rebuild Value (RV), and Refactoring Adjustment (RA). RF refers to the number of lines of code that needs to be modified to improve software quality. RV refers to the effort estimation required to rebuild a system using a particular technology. RA refers to the percentage discount that is made to the repair effort due to context-specific aspects of a project.

$$RE = RF \times RV \times RA \tag{3.1}$$

Interest was quantified as Maintenance Effort (ME) in terms of Maintenance Fraction (MF), Quality Factor (QF), and RV. MF refers to the annual maintenance effort in terms of percentage of lines of code added, changed or deleted, on a yearly basis. QF is a factor used to account for the level of quality.

$$ME = \frac{MF \times RV}{QF} \tag{3.2}$$

The cost of technical debt is broken down into principal, recurring interest (RI) and compounding interest (CI) by Chin et al. [15]. The principal is the cost of servicing the debt; RI is the cost to the organization for not surviving the debt, and CI is the additional technical debt that grows with time.

Curtis et al. [21] focused on the principal calculation as a measure of technical debt, using a function of the number of should-fix violations in the software, the hours to fix each violation, and the cost of labor as illustrated below. Should-fix violations are defined as violations of good coding practices and architectural practices.

$$Principal = ((\sum high - severity\ violations) \times (percentage\ to\ be\ fixed) \times (average\ hours\ needed\ to\ fix) \times (US\ \$\ per\ hour)) + ((\sum medium - severity\ violations) \times (percentage\ to\ be\ fixed) \times (average\ hours\ needed\ to\ fix) \times (US\ \$\ per\ hour)) + ((\sum low - severity\ violations) \times (percentage\ to\ be\ fixed) \times (average\ hours\ needed\ to\ fix) \times (US\ \$\ per\ hour)) \quad (3.3)$$

SonarQube[4] is a common tool for quantifying technical debt (in terms of person days to pay the debt), and the underlying formula is based on the following cost:

$$Debt = cost\ to\ fix\ duplications + cost\ to\ fix\ violations + cost\ to\ comment\ public\ API + cost\ to\ fix\ uncovered\ complexity + cost\ to\ bring\ complexity\ below\ threshold + cost\ to\ cut\ cycles\ at\ package\ level \quad (3.4)$$

where

$$Cost\ to\ fix\ duplications = cost\ to\ fix\ 1\ block \times duplicated\ blocks \quad (3.5)$$

$$Cost\ to\ comment\ public\ API = cost\ to\ comment\ 1\ API \times public\ undocumented\ API \quad (3.6)$$

[4]http://www.sonarqube.org/

$$Cost\ to\ fix\ uncovered\ complexity = cost\ to\ cover\ 1\ complexity \times$$
$$uncovered\ complexity\ by\ tests \quad (3.7)$$

$$Cost\ to\ bring\ complexity\ below\ threshold =$$
$$cost\ to\ split\ 1\ method \times function\ complexity\ distribution \geq 8 +$$
$$cost\ to\ split\ a\ class \times class\ complexity\ distribution \geq 60 \quad (3.8)$$

$$Cost\ to\ cut\ cycles\ at\ package\ level\ =$$
$$cost\ to\ cut\ an\ edge\ between\ 2\ files \times package\ edges\ weight \quad (3.9)$$

There are default values for each cost that can be easily customizable in SonarQube.

Marinescu computed design technical debt as the Flaw Impact Score (FIS) for design flaw instances using three criteria, namely influence (Iflaw type), granularity (Gflaw type), and severity (Sflaw instance). This method measures the impact of design flaws based on how they affect good design in terms of coupling, cohesion, complexity and encapsulation, the level of granularity at which they affect design entities (e.g. class and method) and the severity of the design flaw [58].

$$FIS_{flaw\ Instance} = I_{flaw\ type} \times G_{flaw\ type} \times S_{flaw\ Instance} \quad (3.10)$$

Consequently, the design debt in a system was computed as an overall score - the Debt Symptoms Index (DSI) as follows:

$$A = \frac{\sum all\ flaw\ instances \times FIS_{flaw\ instances}}{KLOC} \tag{3.11}$$

where KLOC is the number of thousands of lines of code of the system.

Ho et al. [37] proposed a method for estimating technical debt (TD), assuming project size is known, based on the COCOMO II model. The effort to pay off technical debt (ETD) is

$$E^{TD} = KLOC^{TDAF} \times EAF \tag{3.12}$$

where TDAF (TD adjustment factor) is the ordered weighted average (OWA) of the 4 dimensions of technical debt: process rules compliance, quality testing, maintainability and complexity, and EAF (effort adjustment factor) are the project and product drivers that will influence how efficiently the debt will be addressed.

$$TDAF = OWA \sum_{i=1}^{4} \sum_{j=1}^{d_i} w_{ij} u_{c_j} \tag{3.13}$$

$$EAF = \prod_{l=1}^{17} EM_i \tag{3.14}$$

(using the 17 cost drivers from COCOMO II)

Singh et al. [78] came up with some preliminary ideas for a framework to estimate technical debt interest based on code and developer comprehension metrics. Code com-

prehension metrics include count class coupled, count class base, count class derived, count line code and count declared method, while developer comprehension metrics include the number of sessions, class visits and other class accesses, and time spent in class and other classes. Therefore, interest is defined as the difference in time spent by the developer to understand the class under the current code structure versus under ideal code structure.

Regarding technical debt measurement, none of the methods described above have been widely adopted by the industry. Many companies are either using internal tools to quantify their debt or not measuring their debts at all. This shows that there are some aspects of technical debt quantification that these methods are lacking, and the lack of standard measurement tools is preventing their extensive implementation. Therefore, further research is needed to curb the gap towards quantification models and techniques that would be a better fit to help companies to measure their debt.

3.3 Decision Making

Seaman et al. [76] borrowed techniques from other disciplines such as finance and psychology to prioritize technical debt items. Such techniques include Cost-Benefit analysis, Analytic Hierarchy Process (AHP), the Portfolio method and the Options approach. The *Cost-Benefit Analysis Approach* uses the principal and interest approach and is better suited in cases where there is limited or no historical data available. A mix of expert opinion and historical data can be used to build the cost-benefit model. *AHP* assigns weights and scales to criteria that are used to measure technical debt and subsequently perform pair-wise comparisons between alternatives to get a prioritized ranking of the technical

debt items. While some of the decisive criteria for technical debt would be principal and interest, human intervention is acceptable in cases where the relevant information cannot be obtained. The *Portfolio Approach* relies on the return on maximization of investment value and investment risk minimization to make informed decisions in technical debt management. However, this approach is specific to the financial domain. For technical debt management, the approach cannot be used as is and needs to be customized. The *Options Approach* is analogous to investing in refactoring the debt item with the long-term objective of facilitating future maintenance and thereby saving money. However, this technique requires the estimation of the key parameters that are difficult to estimate in practice.

Snipes et al. [81] suggested technical debt decision-making factors as severity, existence of a workaround, urgency of the fix required by a customer, effort to implement the fix, risk of the proposed fix, and scope of testing required. They evaluated the factors using semi-structured interviews with 7 ABB control system team members to understand the technical debt management strategy. They found that the decision-making factors are listed in decreasing order of importance with severity being the most influential.

Schmid [74] proposed a formal technique to aid technical debt decision-making. The study distinguished between potential and effective technical debt. Potential technical debt (PTD) is any type of sub-optimal software system (or part of it) while effective technical debt (ETD) are issues in the software system that makes further development of that system more difficult. The approach considers the evolution cost, refactoring cost and probability that the predicted evolution path will be realized in the decision-making process.

Due to limited resources to handle debt, prioritizing debt decisions is critical. A good approach would be to take into consideration a combination of factors that will enable teams to determine how critical the debt item is to the future releases of the software. Such an approach is still nonexistent and current decision-making strategies rely solely on one factor such as customer request or severity of the debt. Therefore, it is important to research this aspect of technical debt further by combining multiple approaches that consider various facets of the problem.

3.4 Impact of Technical Debt

At the start of a new project, it is common to intentionally incur technical debt in order to achieve some goals. When technical debt is incurred for strategic reasons, it is due to the opportunity cost of releasing software now compared to some point in the future. For instance, McConnell points out that when time to market is critical, incurring technical debt might be a good business decision. Other instances where debt can be incurred may be due to time and resource constraints where the software or feature needs to get out of the door and the software will be "fixed" after the release. In addition to time-to-market factor, he also explained how preservation of startup capital contributes towards technical debt. In a startup company, expenses that can be delayed should be as opposed to expending startup funds on technical debt now. Another case where it might be justifiable to incur technical debt would be near the end of a system's lifecycle because when a system retires, all the debts retire with it.[5]

[5] S. McConnell, "Technical Debt," http://www.construx.com/10x_Software_Development/Technical_Debt/ (accessed Apr. 14, 2013)

As explained earlier, technical debt is not always bad if it allows a business to achieve a competitive edge and market share. In such cases, it makes sense to delay the moment when the software is brought in line with standards and best practices. Good technical debt has one or more of the following characteristics [15]:

- It has a low-interest rate - technical debt with low-interest rate can be supported for a longer time frame as the development costs won't increase over time. Repaying the debt at a later time might cost significantly more than it will cost to repay it now.
- It is being regularly serviced - when technical debt is regularly serviced, the amount of debt will decrease over time, thereby increasing the quality of the software.
- The work that preceded it had a very high opportunity cost - if there is a high opportunity cost related to the debt, it is preferable to release the software and incur some debt, rather than not releasing and losing money.

The decision to assume the debt must be made explicit and should be the result of a collective decision based on the key stakeholders concerned. Such a decision will normally be taken after assessment of the risks and benefits identified. In addition, the stakeholders must ensure that a process exists to manage the debt and that it is paid back within a set period of time.

However, non-strategic debt can be detrimental to the quality of the software. Such situations might include when the debt is taken on without stakeholders' approval and the impact of the debt is not properly assessed. It is harder to manage technical debt

accumulated due to shortcuts taken because of poor quality assurances processes.[6] This type of debt is comparable to credit card debt as it can be easily incurred and adds up very fast due to compounding interest.

Nonetheless, a process to pay back technical debt is needed. The longer the debt repayment is deferred, the harder and more costly it will be to pay it back as the interest charges keep compounding. Technical debt is often quantified as principal and interest. These are the basic factors for calculating the Return on Investment (ROI) on resolving the debt if the costs can be determined.

Over time, technical debt can lead to degeneration of the system's architecture. The lack of prompt debt payment can result in technical bankruptcy where an organization's resources are spent dealing with the inefficiencies created by the debt that has accumulated over time and can no longer keep up. In the worst case, the software might need a complete redesign or need to be rewritten; entire departments are outsourced; customers and market shares lost; and customer confidence is lost as the company will be spending more resources on debt servicing rather than focusing on new features [38].

A famous example is Netscape Navigator which experienced architecture decay over a short time period. Netscape developers wanted to release a newer version of the software but could not because the code was harder to change than expected and the system became unmaintainable. The architecture was difficult to understand and it became almost impossible to add new components [33]. Another example is Visual Query Interface (VQI), a software package that degenerated as the programmers made changes to the system without

[6] S. McConnell, "Technical Debt," http://www.construx.com/10x_Software_Development/Technical_Debt/ (accessed Apr. 14, 2013)

following the architectural guidelines provided. The programmers introduced design pattern violations which cause unnecessary couplings, misplaced classes (i.e. classes placed in the wrong package), and imported classes not used in the package [85]. This degeneration is referred to as code decay [38].

Code decay can have several root causes. One cause is violating architectural design principles. For example, in a strictly layered system where a layer can only use the services provided by the layer below. If a developer does not follow the constraint, this change is considered a violation of the architecture.

Other causes for code decay include:

- Time pressure that causes programmers to knowingly postpone refactoring
- Writing code without following proper programming conventions
- Debugging code improperly
- Taking shortcuts to get a working solution as fast as possible

The above examples illustrate that technical debt gives rise to code decay, which makes code changes harder than they should be.

Hochstein et al. [38] reported that in order to find areas of code decay, they used a tool to detect code smells, which helped to identify code areas where good design principles were breaking down. A code smell is a surface indication that usually corresponds to a deeper problem in the system [30]. As a result, code smells are useful to identify areas accumulating technical debt.

In the next subsection, we discuss technical liability, one of the consequences of technical debt that is often overseen in the industry.

3.4.1 Technical Liability

Technical liability[7] is the result of technical debt. This term addresses the cost of business outcomes that arise due to issues with the software product. It is defined as the financial risk exposure and other liabilities over the life of the code. These liabilities have financial implications.[8]

Examples include:

- Future service costs (handling service desk calls for support of hastily released new features to address bugs and customer complaints)
- Fines resulting from privacy violations
- Loss of business from failing a compliance audit
- Loss of intellectual capital due to security flaws
- Loss of customers resulting from a negative reputation for quality

However, technical liability is context dependent (e.g. an automobile software release assumes more liability than the next release of Pokémon Go).

This technical liability concept includes the realization that costs associated with handling technical debt exceed the cost in person-hours to fix. Technical debt does not cover

[7] Self Insurance and Technical Liability, https://www.ibm.com/developerworks/community/blogs/RationalBAO/entry/self_insurance_and_technical_liability?lang=en (accessed Jan. 16, 2014)

[8] Technical Liability: Extending the Technical Debt Metaphor, https://www.ibm.com/developerworks/community/blogs/RationalBAO/entry/technical_liability_extending_the_technical_debt_metaphor?lang=en (accessed Jan. 16, 2014)

the full economic impact of shipping code. A full view of the economic decision to ship must include not only the technical debt, but also the associated technical liability. Technical liability is much more than the typical development and maintenance costs as illustrated in the examples above. Technical liability includes estimates of the possible future costs resulting from the decision to ship. Therefore, before debt ridden code is released, there is a need to think beyond the cost associated with fixing the debt.[9] The software engineering research community often overlooks this broader perspective on technical debt. This work will include technical liability when reasoning about the cost of technical debt.

3.5 Practitioner Studies

Finally, we identified the following empirical studies that reported on software practitioners' involvement with technical debt using empirical methods including interviews and questionnaires. We summarize the studies and their findings.

Klinger et al. [51] report on a case study carried out with four technical architects at IBM to understand the decision-making process used to incur technical debt and determine whether the analogy between technical debt and financial leverage holds. They pointed out that unintentional debt is more problematic than intentional debt. In addition, technical debt decisions were made by non-technical stakeholders and not the technical architects.

Lim et al. [55] describe a study where they interviewed 35 software practitioners to evaluate how the practitioners characterize, perceive and understand the context in which technical debt occurs. Holvitie et al. [39] conducted a survey of 54 practitioners in a Finnish company to understand their perception of technical debt, the effect of agile prac-

[9] Self-insuring your software, https://www.cutter.com/article/self-insuring-your-software-425106 (accessed Aug. 6, 2016)

tices and processes on technical debt and how technical debt manifested itself in projects. They reported that frequent occurrences of technical debt were due to architecture issues, legacy components and increased component size. Yli-Huumo et al. [89] investigated the causes, management and effects of technical debt in a Finnish company by interviewing 12 practitioners. Lim et al. [55] and Holvitie et al. [39] found that most of the participants were unfamiliar with the term technical debt. After being introduced to the concept, most of the practitioners in the study by Lim et al. [55] recognized its existence. The participants pointed out that most of the debts were incurred intentionally, mostly because of customers' inability to provide concise requirements, confirming the findings of Yli-Huumo et al. [89].

Codabux et al. [17] describe a case study with a mid-sized company where they interviewed 27 software practitioners. The aim of the interviews was to understand how practitioners define, characterize, and prioritize technical debt. They reported that developers have their own technical debt taxonomy. In addition, they also pointed out that, despite the company being unaware of the long-term consequences of technical debt, one best practice to manage the debt is to allocate an iteration per potentially shippable increment (PSI) towards reducing the debt. On the other hand, Yli-Huumo et al. [89] pointed out that the participants recognized that technical debt has financial consequences in the long-term but nevertheless, the company does not having any process to handle the debt.

Spinola et al. [82] described a study where 37 software practitioners were asked via an online and paper survey to rank 14 statements on technical debt using a 5-point Likert scale, according to what they mostly agree with. They reported that the participants recognized that technical debt is an important concept in software project management and not just

"bad code". Al Mamun et al. [57] investigated the existence of code smell and root causes of technical debt by carrying out a case study and developers' interviews using self-driving miniature car projects. They pointed out that the root causes of technical debt were debt arising due to time pressure, incomplete refactoring and reuse of legacy, third party, or open source code.

Since the term technical debt was coined about 20 years ago, research has been mostly confined to the basic aspects of the concept. Most studies to date collect and analyze data using static analysis tools and report their findings. Very few studies have been carried out where software practitioners were involved and shared their experiences concerning day-to-day activities dealing with technical debt.

We are aware that empirical studies are a way to curb the gap between theory and practice as it enables researchers to gain knowledge of how software engineers are actually working. Therefore, we need an increasing number of empirical studies to understand practitioner management of technical debt. Our case studies described in Section 4.2.1 adds to the existing literature by carrying out empirical studies with software practitioners to further evaluate practitioner understanding of technical debt and focuses on other important aspects such as risk management and technical liability.

CHAPTER 4

METHODOLOGY

This chapter elaborates on the goal of the dissertation and the research questions as well as gives an overview of the research design and analysis.

Software companies have limited resources in terms of manpower, money, and time. It is common for one person at a company to assume multiple roles. In addition, companies face budget cuts and are expected to achieve more work in the same amount of time. There is also pressure to deliver software quickly due to various strategic reasons, for example, to acquire market share before competitors. Therefore, there is extreme pressure on managers to maximize use of development time to deliver quality software. There is a constant struggle in companies to handle the technical debts that are most risky for the future of the software before the software is released.

Our aim is to address these issues with the following research goal:

To develop a framework that project managers can use in their decision-making process to prioritize technical debt items.

Developers and managers are often faced with the question: which technical debt items are the most critical for the project? Prioritizing technical debt items the wrong way might not be beneficial for the project and might be a misuse of valuable resources (e.g. time and

manpower). This decision-making process is contextual and requires the input of the team members and managers directly involved with the project as they understand the software better than anyone else. For instance, not all technical debt needs to be repaired. A feature or part of the system that is nearing the end of its life can be excluded from the list of technical debt items to be prioritized.

It is crucial to manage technical debt in software projects to avoid disastrous consequences as we explained in Section 3.4. If technical debt is given too little importance, the project will collapse and become unmaintainable. Consequences can include major refactoring or rebuilding. On the other hand, if given too much importance, new features get delayed and the software company's offerings may not be as appealing to potential customers.

Managers tackle the most critical debt items in the time allocated for technical debt management. The decision-making framework outputs a prioritized list of technical debt items from the most critical to the least critical that needs to be addressed. With a prioritized list, managers can decide which debt item is most critical for the software and address them, thus making optimized use of their technical debt management time. In order to build a framework to decide which technical debt item is most critical and needs to be addressed first, we first need to understand how technical debt is perceived among software practitioners.

The overall goal will be addressed using the research questions presented in the next section.

4.1 Research Questions

This section elaborates the research questions.

4.1.1 RQ1: What are the prevalent notions of technical debt?

The rationale behind this question is to understand how the technical debt concept varies among practitioners. We want to investigate whether their definition and technical debt management are influenced by the nature of their work, project, organization or any other factor. In addition to the technical debt taxonomy, we are interested in quantifying technical debt and how technical debt is managed including its impact on the project and how practitioners prioritize risk. We will gather the opinions of practitioners with varying experience and roles in the software development industry on the multiple facets of technical debt. The insights from practitioners will allow us to understand common practices for technical debt management in industry and identify different management factors that we will use in our framework to prioritize technical debt items.

4.1.2 RQ2: What are the different types of technical debt indicators?

Prior to building our framework, we need to identify the different ways that can be used to identify technical debt items. Some companies may already have a list of technical debt items in their backlog that they need to tackle. Therefore, they can use the framework directly on these technical debt items. However, there are other companies who need to identify these debt items prior to using our framework. Our aim is to identify what the potential indicators of debt in the source code are so that we can extract the debt items. In this section, we also elaborate on the relationship between the indicators and problems in the

source code. It is equally crucial to understand which debt indicator can be obtained fairly effortlessly as using the framework to identify the most critical items can be something that some companies do on a regular basis.

4.1.3 RQ3: How do we build a framework to determine the most critical technical debt items?

Our proposed decision-making framework consists of four phases. The first phase is to use the indicators reported in RQ2 to extract the technical debt items. The second phase is to use a prediction model to determine how problematic (i.e. cause additional problems later and is defect prone and risky from a technical liability standpoint) the extracted technical debt items are. Next, we derive a classification scheme from the data to group the technical debt items according to low, medium, and high severity [22]. The last step is to rank the technical debt items using a decision model. We use the debt management factors from our empirical case studies as criteria for ranking the debt items.

4.1.4 RQ4: How effective is the framework?

Once we have developed our framework, we want to determine how effective it is in identifying the critical technical debt items. We use Findbugs[1], a static analysis tool used to determine problems in Java programs by finding instances of bug patterns, to compare our outputs. Findbugs assign their bugs a ranking from 1-20. The categories are scariest (rank 1-4), scary (rank 5-9), troubling (rank 10-14), and of concern (rank 15-20).

[1] http://findbugs.sourceforge.net/

4.2 Research Overview

We conducted three types of studies as shown in Figure 4.1 to uncover answers to the research questions.

- literature reviews which aggregate knowledge about different aspects of technical debt (Section 4.2.1)
- case studies where we interviewed and surveyed software practitioners about technical debt (Section 4.2.2)
- historical data extraction of real software systems to better quantify technical debt (Section 4.3)

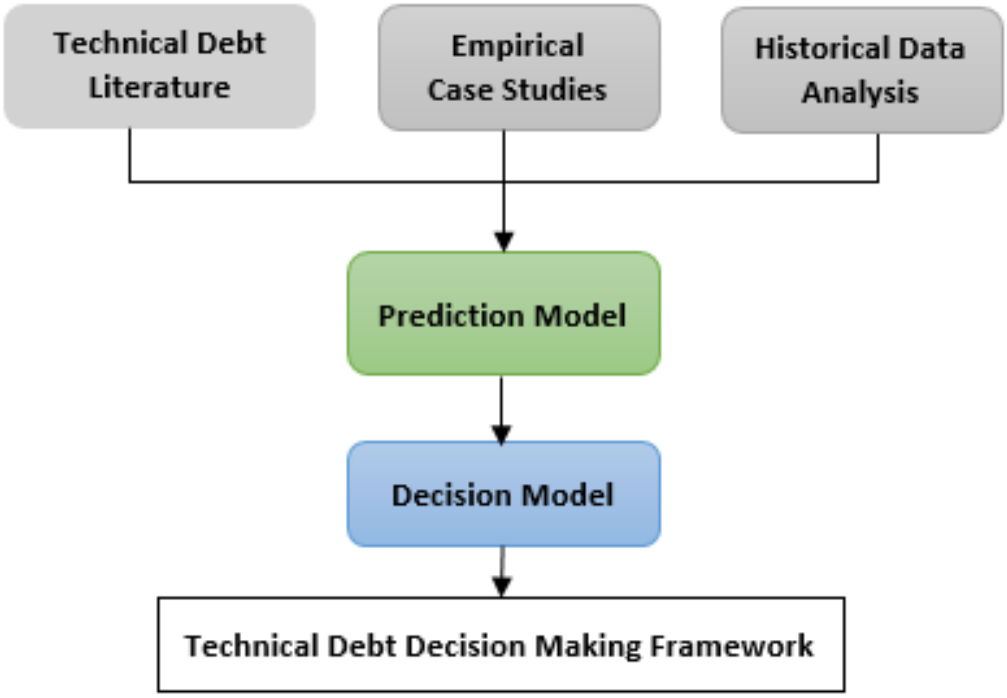

Figure 4.1

Research Overview

4.2.1 Literature Review

First, we conducted a literature review on technical debt to better understand the state of the art. There has been increased awareness of technical debt research since the first Managing Technical Debt workshop in 2010. Most of the studies conducted are on technical debt definitions and taxonomies, quantification, decision-making, impact, and empirical studies involving practitioners. These studies have been reported in Chapter 3.

Next, to understand the different technical debt indicators, we surveyed the literature to extract the different types of indicators reported in the academic literature, and we also investigated whether these indicators indicate problems in the source code.

4.2.2 Empirical Case Studies

In 2012 when we started studying the technical debt metaphor, there was very little academic research on technical debt. Mostly, practitioners were talking about technical debt on their blogs. Therefore, we also conducted a case study with an industrial partner [17], a leading global provider of networking, and communications equipment to examine some aspects of technical debt that were pertinent to this research. We used semi-structured interviews, and a questionnaire to gather data for this study. We conducted 30-minutes semi-structured interviews with 27 practitioners to assess their viewpoints of technical debt (both management and technical) and to understand the terminologies used. The interviews were of both individual developers and Scrum teams (6-9 team members). We used a questionnaire to obtain background information (e.g. work experience) on each interview participant. Participation in the interviews was voluntary. The potential participants

included all engineers within the division. Our industrial partner's management requested volunteers to give their feedback and many obliged. The interviews were conducted over a two-day period from 9-5 P.M.

The second case study [19] was conducted in 2014 and was carried out in two phases as illustrated in Figure 4.2. The aim of this study is to understand the state of technical debt (taxonomy, management, communication, impact, etc) and to understand debt risk assessment from a practitioner's point of view. The first phase consisted of semi-structured interviews with software practitioners. We interviewed 17 practitioners from different geographical locations worldwide. The interviews were conducted over the phone over a three month period. Each interview was roughly 30 minutes with some interviews lasting longer. While we had a comprehensive list of questions, most of the time, we did not feel the need to ask all of them as the participants provided answers while addressing other questions. Thus, most interviews were concluded within the 30-minute timeframe. The participants were recruited using a convenience sample of professional contacts. Some interviewees also provided contact information for other potential participants at different companies. The sample included practitioners with varying years of experience working with a range of different software products. Four of the participants were software developers. The remaining participants consisted of 2 consultants, 3 technical leads, 3 architects, and 4 in management positions. The remaining participant had a research position. The interview questions comprised both open-ended and closed-ended questions. In addition, demographic questions were sent in advance to be completed prior to the interview.

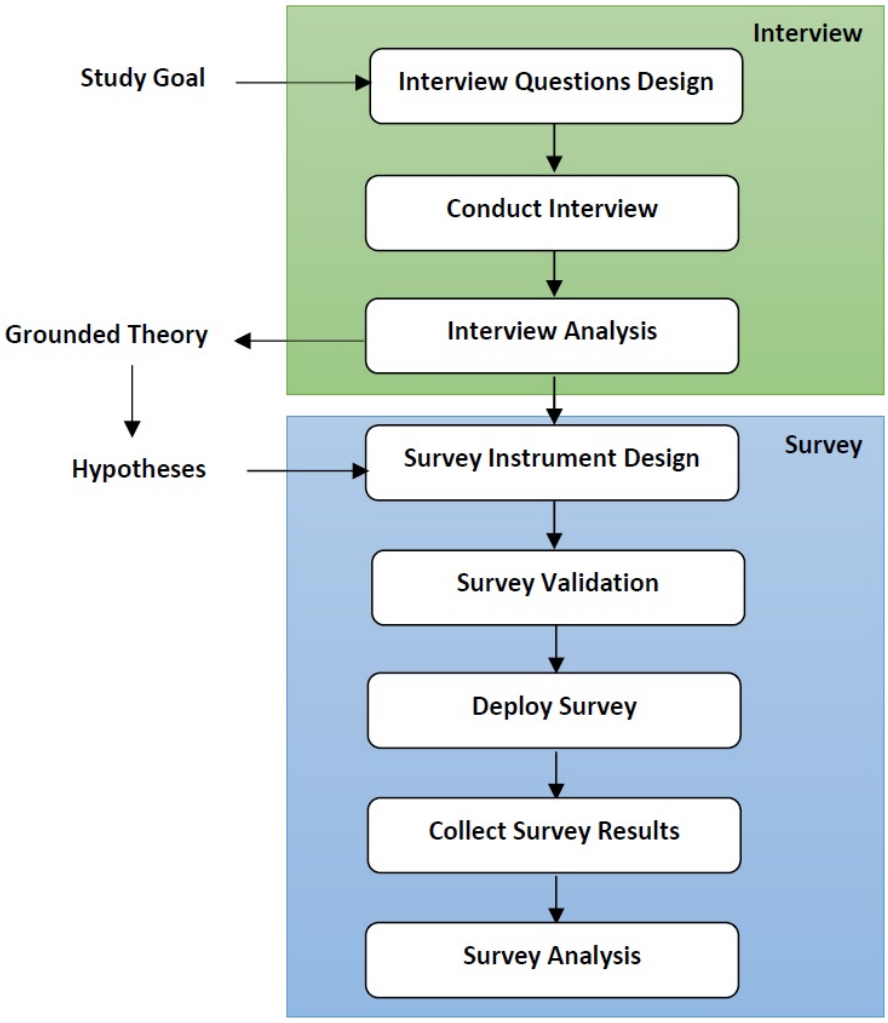

Figure 4.2

Case Study Design

To analyze the interview data, we used a Grounded-Theory approach [32]. Following the interviews, we coded the data (i.e. we generated some technical debt themes from the data) based on our goal. Next, we generated our hypotheses on technical debt from these themes. Lastly, we translated these hypotheses into survey questions.

For the second phase, as described above, we used the insights gathered from the interviews to design a survey instrument which we deployed to the software engineering community. Participation in the survey was on a voluntary basis, and participants were recruited by asking the following Software and Systems Process Improvement Network (SPIN) chapters' administrators to send the survey to their mailing lists:

- SPIN chapters in the US (http://www.sei.cmu.edu/spin/find/us/index.cfm)
- International SPIN chapters (http://www.sei.cmu.edu/spin/find/international/index.cfm)

In addition, the survey was advertised on:

- Ontechnical debt blog (http://www.ontechnicaldebt.com/blog/technical-debt-study/)
- LinkedIn groups related to technical debt and agile (Tech Debt, Technical Debt, Agile, Agile and Lean Software Development, Disciplined Agile Delivery, Agile Application Lifecycle Management, Certified Software Development Professional ((CSDP)) groups).

The survey invitation was also sent through email to the authors' professional contacts.

4.3 Historical Data

This section describes the details of the historical data extraction. We describe the systems studied, the tools used for the data extraction and the data extraction process as well as the process for building the framework.

4.3.1 Corpora

We selected two open-source software projects namely Apache Hive and Apache Mahout from the Apache Software Foundation using the following criteria:

1. Projects are written in Java
2. Source code is publicly accessible
3. Project issues are publicly accessible
4. Source code is managed using SVN
5. Project issues are managed using JIRA
6. Project should have multiple release versions
7. Project should have at least 1000 issues

4.3.1.1 Apache Hive

Apache Hive[2] is a large-sized open source project with more than 900K source lines of code (SLOC) and is part of the Apache Software Foundation projects[3]. It is a data warehouse infrastructure built on top of Hadoop that facilitates querying and managing large datasets residing in distributed storage. Hive provides a mechanism to project the structure onto this data and query the data using an SQL-like language called HiveQL.

[2] Apache Hive, https://hive.apache.org/ (accessed May 1, 2015)
[3] Projects Directory, https://projects.apache.org/ (accessed May 1, 2015)

It should be noted that we ignored early versions of the system when it was incorporated in another system. For example, Apache Hive was part of Apache Hadoop prior to version 0.6.0. Therefore, the initial version of Apache Hive that we used was version 0.6.0. Table 4.1 lists the different versions of Apache Hive that we considered in this dissertation and their respective release dates.

Table 4.1

Apache Hive - Versions

Version	Release Date
0.6.0	Nov 3, 2010
0.7.0	Mar 29, 2011
0.7.1	Jun 22, 2011
0.8.0	Dec 19, 2011
0.8.1	Feb 6, 2012
0.9.0	Apr 30, 2012
0.10.0	Jan 11, 2013
0.11.0	May 15, 2013
0.12.0	Oct 15, 2013
0.13.0	Apr 21, 2014
0.13.1	Jun 6, 2014
0.14.0	Nov 12, 2014
1.0.0	Feb 4, 2015
1.0.1	May 21, 2015
1.1.0	Mar 8, 2015
1.1.1	May 21, 2015
1.2.0	May 18, 2015
1.2.1	Jun 27, 2015
2.0.0	Feb 15, 2016

4.3.1.2 Apache Mahout

Apache Mahout[4] is a medium-sized open source project with about 100K SLOC and is also part of the Apache Software Foundation projects. It is a Java-based framework consisting of many machine learning algorithm implementations. The project currently has map-reduce enabled (via Apache Hadoop) implementations of several clustering algorithms (k-Means, Mean-Shift, Fuzzy k-Means, Dirichlet, Canopy, Naïve Bayes and Complementary Naïve Bayes classifiers), and collaborative filtering, as well as support for distributed evolutionary computing.

For Apache Mahout, we consider the major versions including the first and last versions. Table 4.2 enumerates the different versions of Apache Mahout that we considered in this dissertation.

Table 4.2

Apache Mahout - Versions

Version	Release Date
0.1	Apr 7, 2009
0.5	Jun 2, 2011
0.10	Apr 11, 2015
0.12.2	Jun 13, 2016

[4] Apache Mahout, http://mahout.apache.org/ (accessed May 1, 2015)

The characteristics of both systems are summarized in Table 4.3.

Table 4.3

Overview of Analyzed Systems

	Apache Hive	**Apache Mahout**
Revisions	3419	1413
Classes (last version)	11234	1801
SLOC (last version)	904425	109079

4.3.2 Tools

In this section, we describe the tools that we will use for the data extraction and for collecting the metrics.

4.3.2.1 Change Count Extraction Tool

Change count was extracted using the Change Calculator tool, an internally developed software. The Change Calculator is a Java-based tool written to count the number of times a class was changed. The count was based on the number of times a change for a class was reflected in the revision numbers of an SVN repository branch. We extracted change count for each unique class from the first to the last versions identified in Table 4.1 and Table 4.2.

4.3.2.2 Defect Count Extraction Tool

Defect count was extracted using an extension of the JIRA Extractor tool [26], an internally developed software. The JIRA Extractor is a Java-based tool initially written to extract software project issues from the JIRA issue tracking system as well as to extract SVN repository data for revisions associated with JIRA issues. The tool extracts issues from JIRA and gets a list of all the classes from SVN that were changed to address the issue. This tool was extended to include functionality to enable the extraction of defect count for each unique class from the first to the last versions identified in Table 4.1 and Table 4.2.

4.3.2.3 Scitool Understand

Scitool Understand[5] is a commercial static analysis tool for maintaining, measuring, and analyzing source code. We are using Understand solely for the purpose of extracting standard metrics and exporting them to spreadsheets. We are mostly interested in the metrics extracted at the class level. Scitool Understand extracts about 50 metrics. Most of the metrics can be grouped as complexity, volume, or object-oriented metrics.

4.3.2.4 Intooitus inCode

Intooitus inCode[6] is a commercial tool that extracts code smells and architecture violations (referred to as design flaws) in the source code. inCode detects code smells and design flaws by evaluating different metrics. This tool by Marinescu [58] has been elaborated in Chapter 3. inCode detects more than 10 code smells that are most commonly encountered

[5] Understand Static Code Analysis Tool, https://scitools.com/ (accessed May 5, 2015)

[6] inCode, https://www.intooitus.com , its evolution at http://www.aireviewer.com (accessed May 5, 2015)

in software projects such as god class, data class, code duplication, schizophrenic class, and so on. We briefly describe these code smells below:

- Code Duplication refers to groups of operations which contain identical or slightly adapted code fragments. By breaking the essential Don't Repeat Yourself (DRY) rule, duplicated code multiplies the maintenance effort, including the management of changes and bug-fixes. Moreover, the code base gets bloated. inCode identifies three types of code duplication: (i) internal duplication involving methods that belong to the same scope (class or module); (ii) sibling duplication of methods from the same class hierarchy; and (iii) external duplication that refers to unrelated operations.

- God Class is an excessively complex class which gathers too much and non-cohesive functionality and heavily manipulates data members from other classes. This means that the data members and the methods that manipulate them are separated. Such a class is harmful because it indicates that the functionality is improperly partitioned.

- Data Class refers to a class with an interface that exposes data members instead of providing any substantial functionality. This non-encapsulated data is usually manipulated by other classes in the system. This means that related data and behavior are not in the same scope, which indicates a poor data-functionality proximity. By allowing other classes to access its internal data, Data Classes contribute to a design that is fragile and hard to maintain.

- Schizophrenic Class describes a class with a large and non-cohesive interface. The lack of cohesion is revealed by the several disjoint sets of public methods that are

used by disjoint sets of client classes. Such classes represent more than a single key abstraction and this affects the ability to understand and change in isolation the various abstractions embedded in the class.

From now onwards, a class with code smells will be referred to as a technical debt item (TDI).

4.3.2.5 AgenaRisk

AgenaRisk is a powerful tool for modeling, analyzing and predicting risk. AgenaRisk supports predictive reasoning using risk maps (most commonly known as Bayesian Networks) and also allow to visualize risks and their relationships. We used AgenaRisk to generate our Bayesian Network(BN).

4.3.3 Data Extraction

In this section, we will describe the process to extract metrics to build our decision-making framework. This process is illustrated in Figure 4.3.

4.3.3.1 Extract Source code

For the identified versions of the software, we downloaded their source code from https://archive.apache.org/dist/ and also collected release date information for the different versions from their respective official mailing lists.

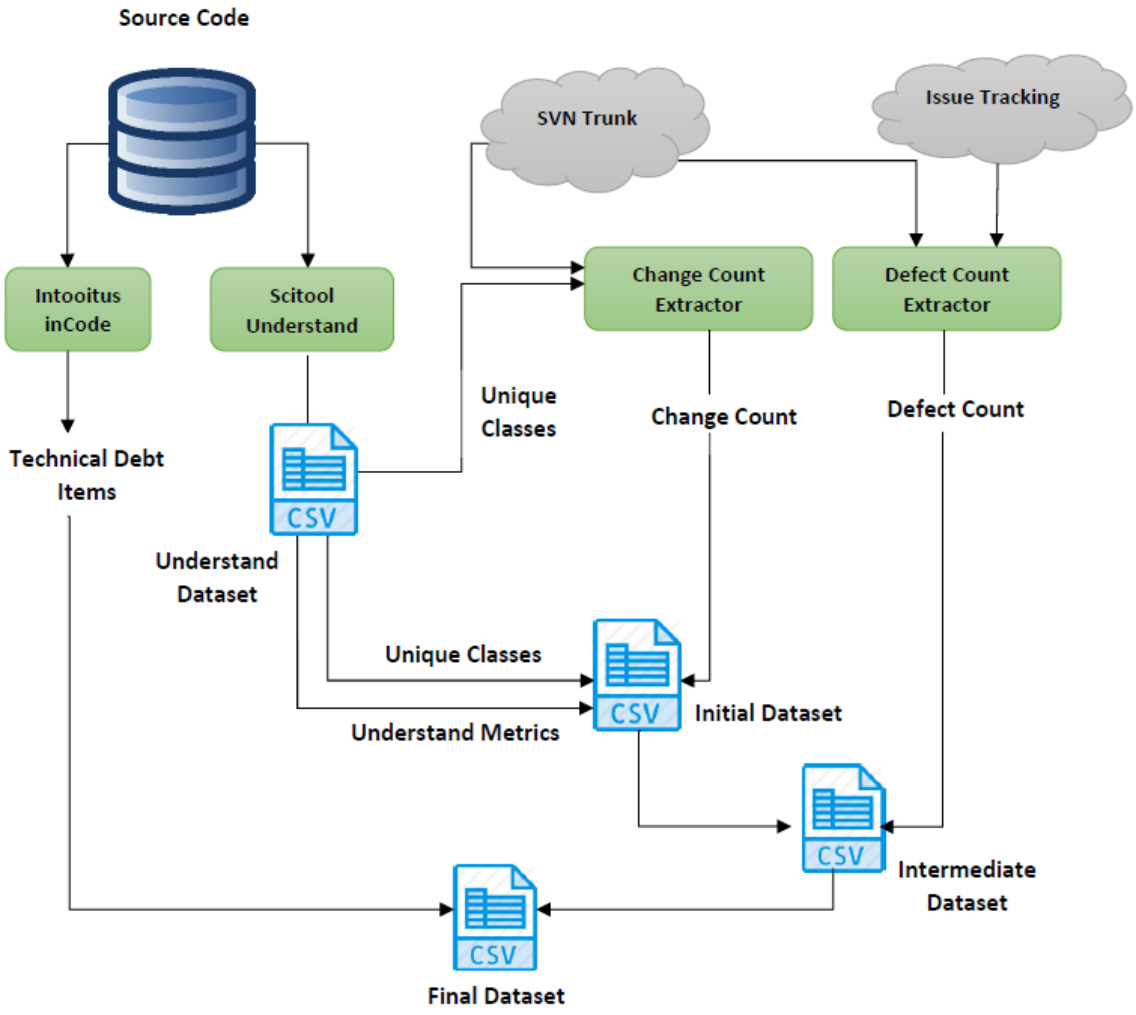

Figure 4.3

Research Overview

4.3.3.2 Collect Understand Metrics

Next, we ran Understand on the "/src" folders of the downloaded source code to get the understand dataset including different metrics. Understand generates about 50 metrics from the source code. We eliminated redundant metrics from the same family (e.g. Sum Cyclomatic complexity, Sum Modified Cyclomatic complexity, Sum Strict Cyclomatic complexity, etc). Then, we extracted a unique set of classes for the identified versions of the software from the Understand dataset using a Java program.

4.3.3.3 Collect Change Count Metric

After having completed these pre-requisite steps, we ran the change calculator. The change calculator extracts the change count information for each class unique set of classes using the SVN trunk.

4.3.3.4 Create Initial Dataset

We combine the unique classes, the Understand metrics and the change count information to get an initial dataset.

4.3.3.5 Collect Defect Count Metric

Next, we ran Jira Extractor to get the list of defect classes in the identified software versions along with the defect count.

4.3.3.6 Create Intermediate dataset

We used a Java program to aggregate the initial dataset with the defect count information, resulting in the intermediate dataset.

4.3.3.7 Collect Technical Debt Count Metric

Next, we ran inCode on the source code to extract the technical debt instances at class level.

Based on previous empirical studies, we extracted technical debt at the class level using a combination of code smells, defect count, and code churn (change count). Code smells indicate problems in the source code. However, not all code smells are relevant to every system. In addition, using static code metrics alone may ignore some causes of technical debt such as poor requirements or design, unexperienced developers, bad documentation, or managerial issues. All of these factors can impact how problematic a system can be, but the code metrics generated by Understand do not consider them effective. Therefore, as a further indication of a problem in the source code, we also extract defect and change count information to help us further narrow down the items that may contain technical debt.

Classes are the building blocks of object-oriented systems and are self-contained elements from the point of view of design and implementation. Defining TDIs depends on the company's business objectives and information that is available to extract them.

We extracted metrics and TDIs from the latest version of the software. Over the lifetime of the class, the most recurring and difficult to remove technical debt are the ones remaining in the project. The trivial and less time-consuming ones are removed more easily than the hardcore debt items which usually require the most effort and resources. As the aim of this framework is to rank the riskiest technical debt items mimicking a setting as close as possible to real-life technical debt management, we use the metrics and identified technical debt items associated with the latest version of each system. When the class is first created,

it may not contain technical debt. If the class was created using shortcuts, it will contain some technical debt, or it might have accumulated debt during its lifetime that will not be visible in the version the class was created. In industry, technical debt is normally tackled when the source code has become too problematic and is becoming harder to maintain; thereby justifying our decision to use the latest version for the technical debt items and associated metrics.

4.3.3.8 Create Final Dataset

The technical debt information from Step 7 was combined with the intermediate dataset, resulting in the final dataset. Prior to finalizing the dataset, we cleaned it to remove any anomalies. For instance, defect data could not be extracted when information such as revision number was missing in Jira for different issues and therefore, was eliminated. In addition, we also performed a stepwise regression [69] on the final dataset to identify the Understand metrics that were significant to be included in our prediction model.

Lastly, for each variable in the final dataset, we identify recommended and non-recommended thresholds based on previous studies in the literature [29, 9, 12] as shown in Table 4.4. For example, a class with a WMC value less or equal to 15 would be categorized as *recommended* and a class with WMC value greater than 15 would be categorized as *non-recommended*. For defect count, change count and TDI count, any class with value 2 and above were flagged as not recommended. We add one additional column for each variable in the final dataset where a *1* would indicate the metric is within the recommended threshold and a *0* would indicate that the metric is not within the recommended threshold.

Table 4.4

Variables' Threshold

Variables	Threshold
LOC	200
WMC	15
CBO	15
LCOM	20
NOC	2

4.4 Decision-Making Framework Construction

After the data extraction phase, we put together our decision-making framework. First, for the identified software version, we extract the code smells that we selected as technical indicators, defect count, and change count as well as other traditional and object-oriented metrics. Next, we use predictive analytics to determine how problematic each of the technical debt items are. Using the probability of the technical debt items being problematic from the prediction model, we categorize the TD items using a classification scheme. After the debt items have been classified, a subjective decision is taken by the project manager or developers to determine which items to consider. This decision is based on the team's knowledge of the software and customer expectations. For example, the technical debt items classified as low and medium, signifying low and medium risk items respectively, can turn out to be part of features which is of high value in the next release of the software.

Similarly, the debt items that fall in the high category can turn out to be in features not so important for the next release and can be addressed later. In the last phase, we build a decision model that will be used to rank the debt items. Typical inputs to the model will be the output from the prediction model, the impact of TD item (i.e. developers' assessment of the debt item based on perceived cost, effort, or business objectives of the system), the location in source code, dependencies between the TD item and other important system functions and code churn (how often the code changes over time). As described in Section 3.3, Seaman et al. [76] propose 4 cost-benefit models for technical debt decision making. In our case, we will rank our debt items using Analytic Hierarchy Process (AHP). The inputs will be the criteria that the team will agree on. The next subsections describe the process of building the prediction model, defining the classification scheme, and constructing the decision model.

4.4.1 Prediction Model

To build the prediction model for technical debt, we use a Bayesian Network (BN) [36, 24, 63, 13]. A BN is a graphical model that shows the probabilistic causal or influential relationships among a set of variables. It is represented as a directed acyclic graph (DAG) with nodes for variables and edges for directed relationships between the variables. For each variable, there is a corresponding node probability table (NPT) which is usually derived from the observed data. NPTs define the relationships and uncertainty of the variables. NPTs specify how the probability of each state of the variable depends on the states of its parents. The variables are usually discrete with a fixed number of states (e.g. recom-

mended and non-recommended) in our case. For each state, the probability that the variable is in that state is given. The BN represents the complete joint probability distribution, that is, assigning a probability to each combination of states of all the variables.

Formally, the relation between the two nodes is based on Bayes' Theorem [83]:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{4.1}$$

where [P(X|Y)] is the conditional probability of the event X given the event Y, and P(X) and P(Y) are the probabilities of events X and Y respectively. Each discrete variable node has an N x M node probability table (NPT) where N is the number of node states and M is the product of its cause-nodes states. In this table, each column represents a conditional probability distribution, and, consequently, its values sum up to 1.

BNs are usually built using a mixture of data and expert judgments. Understanding cause and effect is a basic form of human knowledge underlying our decisions. The expert's understanding of cause and effect is used to connect the variables of the net with arcs drawn from cause to effect. To ensure that our model is consistent with these empirical findings, the probability tables in the BN are constructed using data whenever it is available.

One of the greatest strengths of BNs is allowing probabilistic beliefs about the connections between variables to be updated automatically as new information becomes available, leading to more accurate predictions compared to traditional statistical approaches like regression models. BNs do not only reflect relationships between variables like regression models, but they also direct cause and effect relationships. BNs can also handle incom-

plete datasets and prevent data overfitting [27]. One particularly interesting feature of BNs is the ability to perform a what-if analysis to explore the impact of changes in some nodes to other nodes.

In the field of software engineering, BNs are commonly used for modeling uncertainties in software testing, in defect density prediction, and in other areas [70, 71, 3, 28, 66].

The process of building a BN contains the identification of interesting variables that shall be modeled, representing them as nodes, constructing the topology and constructing the NPTs. We used stepwise regression to clean up the initial variable set and generate the relevant variables. We then define the causal relationships among the variables and generate the probability distribution of each variable. This process is illustrated in Figure 4.4.

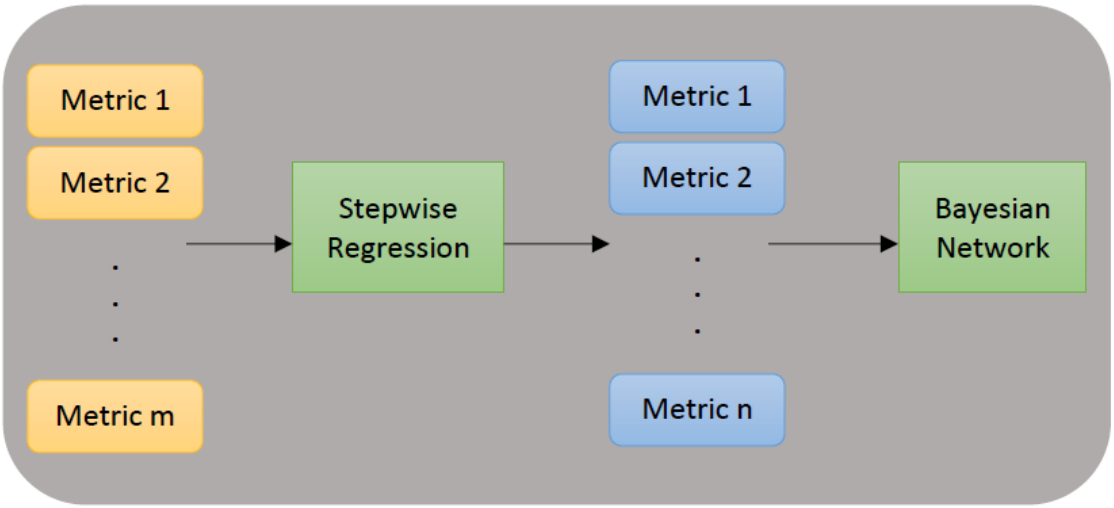

Figure 4.4

Prediction Model Generation Process

We used AgenaRisk to input the conditional probability distribution for each node (independent variable) and generate the Bayesian Network (BN). After building the BN, we simulate different scenarios based on our dataset and observe the respective resulting values for "Technical Debt" (the root node of the BN) being true. Basically, a scenario involves additional information to the model, more specifically, adding an observation to a node. Scenarios can be formulated as what if questions. For example, a typical scenario can be "what if CBO, LOC, WMC, LCOM, Change count and Defect count were all within the recommended threshold." As a result, the probability to fulfill our goal (i.e. the technical debt of a class being true) will be generated. To check the validity of our model, we performed a sensitivity analysis using AgenaRisk. A sensitivity analysis allows us to see which nodes have the greatest impact on the (target) node.

4.4.2 Classification Scheme

Based on the resulting probability of technical debt being true from the BN, a classification scheme is generated from the dataset. For example, a sample classification scheme can be as follows:

- Low risk: Probability lying between 0.0 (inclusive) and 0.4 (exclusive)
- Medium risk: Probability lying between 0.4 (inclusive) and 0.7 (exclusive)
- High risk: Probability lying between 0.7 and 1.0 (inclusive)

At this stage, a subjective decision is made by the project manager or developers to determine which TDIs to consider. For example, a sample selection might consist of 10% low-risk debt items, 25% medium risk debt items and 65% high-risk items.

After the classification scheme has been applied, we further reduce the dataset to about 5 or 10 TDI. There are multiple ways to narrow down the dataset. For instance,

- Selecting the top 5 or 10 TDI with the highest probability (obtained from the BN)
- Selecting the TDI using the highest defect count
- Selecting the TDI based on impact of the debt (highest coupling)

4.4.3 Decision Model

The last step of the framework is the ranking of the TDIs using a decision model. The selected technique is Analytic Hierarchy Process (AHP). AHP is a widely used decision-making approach which uses pairwise comparisons and weightage, resulting in numerical priorities [73].

The rationale for choosing AHP over more traditional decision-making techniques such as cost-benefit analysis is the ability of AHP to adopt a multi-criteria approach which allows us to take into consideration many different factors when ranking the TDIs. Similarly, when project managers or team leaders in the industry need to consider which TDI to tackle, they have to consider various factors related to that particular debt item. Therefore, a multi-criteria approach as part of our framework is most closely related to industry setting for technical debt management.

AHP can be summarized as follows:

1. The problem is modeled as a hierarchy containing the decision goal, the alternatives for reaching it, and the criteria of evaluation.
2. The elements of each level of the hierarchy are rated using pairwise comparisons.
3. Priorities among the elements are established subsequently.
4. Next, we calculate the priorities of the alternatives with respect to the goal.
5. Lastly, we make the final decision based on the results.

In our case, the goal is to find the most critical (problematic) debt item. The criteria are context-dependent and the final selection of criteria is a collective decision of the software team, based on the business objectives.

Potential criteria for AHP are:

- Debt probability (output of the prediction model)
- Impact of the debt (coupling)
- Location of the debt (how critical is that class to the software? is that class part of an important functionality?)
- Age of the debt (the longer the debt stays in the system, the more interest it accrues)
- Code churn (how frequently does this class change?)
- Defect count (is the class problematic?)
- TDI count (a class with hundreds of TDI versus a class having a couple of TDI is definitely a red flag)

- Type of debt (god class or formatting issue or module with no test coverage have different consequences)
- Likelihood that users will interact with the debt items (how heavily is the class where the debt is used? - does everyone interact with it or only a few users interact with the module containing the debt)

CHAPTER 5

RESULTS

This chapter presents the results for the research questions. We elaborate on the notions and identification of technical debt as well as customize and evaluate the framework for both systems.

5.1 RQ1: What are the prevalent notions of technical debt?

We will provide insights for this research question based on our case studies described in Chapter 4. The interview questions, hypotheses, and survey that form the basis of these findings are reported in Appendix A and Appendix B. We have summarized the findings grouped according to different categories.

5.1.1 Definition of Technical Debt

According to the different taxonomies of technical debt presented in Section 3.1, the participants in the industrial case study described both intentional and unintentional debts. In addition, management and developers' definitions of technical debt were different. In the survey, the participants did not limit the definition or description of technical debt. They explained the rationale behind incurring debt, the ways debt was incurred unintentionally, and the advantages of incurring technical debt. This discrepancy indicates that

there is no single definition that characterizes technical debt. The definition and description are conceptualized according to the practitioner's situation and past experiences. This difference in definitions confirms the findings of a case study carried out in one software development division reported by Codabux et al. [17] where the definition of technical debt varied according to the role of the participant.

5.1.2 Impact of Technical Debt

According to the participants in the industrial case study, technical debt was incurred to complete objectives and satisfy the customer. Developers feared to acquire more work if resolving some technical debts meant having to touch other related areas as a consequence of addressing the debt. Addressing the debt could thereby break other features and require more work from the developers. There was only minimal insight obtained regarding the longer-term impact of technical debt. Some practitioners feared that technical debt over the long-term has the potential to seriously hinder a project. Despite the potentially serious consequences of long-term technical debt, practitioners are willing to take on debt to satisfy their short-term requirements. Some impacts of technical debt, according to participants of the survey, were increased time and effort handling code where debt is present. Trivial development tasks become more complicated in areas of the system where technical debt is encountered. Consequently, this affects the velocity of the entire team. When the performance of the software is negatively affected, customer satisfaction decreases. Team members take more time handling debt-ridden tasks. The company is also impacted as technical debt causes an increase in maintenance cost, and low-quality software can

cause potential damage to the reputation of the company. In short, technical debt has a ripple effect - when incurred by a developer, it affects every stakeholder, either directly or indirectly.

5.1.3 Technical Debt Communication

Most participants of the survey mentioned that while technical debt is mostly communicated amongst development team members and management, it is very rarely discussed with clients. Sometimes, management will receive automatic visibility via tools such as SonarQube. In cases where the technical debt was communicated with the clients, the latter were involved with internal technical debt decisions. Communicating debt to the customers assists the development team in prioritizing debt and allows them to focus on features important to the customer. Doing so also helps to mitigate future problems that might arise if the customer discovered the debt during normal operations of the software.

5.1.4 Technical Debt Quantification

Several participants from the survey mentioned that commercial tools such as SonarQube[1] and SQALE [52] were used in their company to quantify debt. Others pointed out that they use custom built tools for debt measurement. Several participants mentioned that technical debt was not quantified at all in their companies. In addition, some of the metrics to quantify debt that the participants mentioned are story points (an indicator of effort necessary to address the debt) and time. This denotes that, despite technical debt being quantified using different techniques, tools, and metrics, money is the "universal

[1] SonarQube http://www.sonarqube.org/ (accessed April 8, 2012)

language" to which it is eventually translated for easier understanding by all stakeholders involved. Regarding technical liability, 60% of the practitioners explicitly agreed that the cost of technical debt is more than the cost of addressing code deficiencies revealed by tools. Some practitioners even mentioned after listening to the interview question that the concept of technical liability is new to them and that after-delivery cost offers a new perspective that they had not considered.

5.1.5 Technical Debt Management

The participants from the industrial case study mentioned three techniques to address architecture / design debt, namely refactoring, repackaging and reengineering. While refactoring is a common technique mentioned in literature [30], reengineering and repackaging are less common. Despite the primary focus of companies being to produce new features, dedicated teams are assigned to reduce technical debt, and a majority of teams spend roughly 20% time in each PSI towards debt reduction.

5.1.6 Technical Debt Decision-Making

Severity and customer impact were considered important when deciding development priorities. Participants, however, did not mention any assessments based on expected effort to address the debt or any risks associated with a fix or the scope of testing.

5.1.7 Technical Debt Risk Management

For about half the participants in the survey, risk related to incurring technical debt is not taken into consideration. A few mentioned that risk is discussed informally. The risks of technical debt are primarily unknown for most practitioners.

In addition to the above insights, the survey revealed distinct company profiles with respect to their approach to technical debt management:

- Companies that use technical debt as a way to manage the iron triangle. The iron triangle[2] is a model of constraints in project management. These constraints typically include scope (features and quality), time, and cost. Such companies manage and quantify their technical debt and understand that the cost of technical debt is more than the effort to fix the code.

- Companies that think of technical debt as a way to manage the iron triangle but do not manage and quantify their debt and view technical debt cost as the effort associated with handling the debt.

- Companies that use technical debt to reason about the cost of fixing deficiencies left in the code after it has shipped out but do not manage and quantify their debt and view technical debt cost as the effort associated with handling the debt.

5.2 RQ2: What are the different types of technical debt indicators?

The way technical debt is identified can be very informal (e.g. developers noting down "pain points" that slow their velocity on a sticky note) and a difficult task, often depen-

[2]Project Management Triangle, https://en.wikipedia.org/wiki/Project_management_triangle (accessed October 11, 2015)

dent on the resources are currently available (e.g. companies often prefer to use existing tools to identify technical debt rather than investing in new ones). More formally, the main technical debt indicators reported in the literature are modularity violations, design pattern grime, code smells, and antipatterns [90]. We give an overview of each on the next subsections and also report empirical studies which have been conducted to determine how these different indicators can be used to identify problems in software.

5.2.1 Modularity Violations

Modular software supports the separation of the functionality of the software into independent distinct modules. However, due to quick and dirty implementations or inability of architecture to adapt to changing requirements, modules may change together instead of being independent. This results in modularity decays and, consequently, necessitates expensive reworks [88]. However, modularity violations are hard to detect using existing tools as they do not influence the functionality of the software directly.

The study by Wong et al. [88] describes CLIO, an approach that detects and locates modularity violations. The workings of CLIO are based on comparing how components should change together based on the modular structure and how they actually change together based on revision history. CLIO was evaluated using 15 versions of Apache Hadoop and 10 versions of Eclipse JDT. CLIO identified 231 violations, and 66% of these violations were confirmed based on the fact that these violations were indeed addressed in later versions or were flagged as potential problems by programmers in source code comments. Similarly, CLIO identified 399 violations out of which 40% were confirmed.

5.2.2 Design Pattern Grime

Design patterns are popular, reusable best practices for successful software design and architecture in the form of templates and descriptions. They promote software maintenance and improve documentation [86]. Grime is the buildup of unrelated artifacts (non-pattern code) in classes that play roles in a design pattern realization. These artifacts do not contribute to the intended role of a design pattern. Grime is observed in the environment surrounding the realization of a pattern [40]. Grime is a component of design disharmony and an indicator of technical debt [90].

Dale et al. [23] investigated the effect of grime on technical debt. They designed a grime-injector that modifies bytecode with couplings in students' Java projects to model six types of design pattern growths. Then, they used SonarQube to calculate the technical debt scores for these projects. They reported that projects with temporary grime have statistically higher significant technical debt scores than projects with persistent grime. However, the experiments were conducted on programs from an introductory software engineering class and not on real industry projects. Second, the grime-injector is far from being a "grime-detector," which would have been more appropriate for identifying technical debt.

Izurieta et al. [40] reported the effects of grime on code decay. A case study was conducted with the JRefactory Open Source System to determine how design patterns decay as the system evolves. In addition to finding out that some design patterns have more grime than others, one of the findings of the study was that as patterns evolve, grime buildup increases. These results were confirmed by an exploratory multiple case study by the same

authors [42]. Grime is one of the indicators of technical debt, and technical debt causes code to decay as we pointed out in Chapter 3.

Izurieta et al. [41] reported the effects of grime on testability. Based on a case study conducted with the JRefactory Open Source System, they reported that as the systems age and grime builds up, the software requires more testing. This clearly indicates that as technical debt increases in a system, the systems becomes more problematic and require to be tested more to maintain its quality.

5.2.3 Code Smells

A code smell is a surface indication that usually corresponds to a deeper problem in the system [31]. As a result, code smells are useful to identify areas accumulating technical debt and need refactoring. Fowler et al. [31] introduce 22 bad code smells. Commonly studied code smells are duplicated code, feature envy, refused bequest, data class, long method and large (god) class [93].

Compared to the other technical debt indicators, code smells are the most studied. Next, we provide an overview of a few studies on code smells and their effect on code maintainability.

Regarding the code duplication code smell, Monden et al. [61] investigated the effect of code duplication on software maintainability using 20-year old industrial legacy system data. They concluded that code duplication reduces software maintainability, supporting Fowler's claim regarding the code duplication [30]. Zhang [92] studied the relationship between duplicated code, data clumps, switch statements, speculative generality, message

chains, and middle man code smells and defects in 5 versions of Eclipse and 6 versions of Apache Commons. They reported that code duplication is more likely to be associated with defects than the other types of code smells and should be considered highest priority when refactoring the code. To determine to what extent code duplication affects change-proneness, Lozano et al. [56] studied four open source systems. They concluded that change effort increases when methods are subject to code duplication. Kapser et al. [46] studied the impact of 11 types of code duplication on the medium open source systems Apache httpd and Gnumeric. They concluded that not all types of code duplication were harmful to the systems. A study by Juergens et al. [45] on five projects, including one open source system, reported that for Java and C++ code, the inconsistently changed duplicated code contained more defects than average code. For COBOL, the inconsistently changed duplicated code did not have much effect on the code.

A study carried out by Olbrich et al. [67] on god class and classes with shotgun surgery code smell analyzing the historical data of Apache Lucene and Apache Xerces 2 J showed that the classes with code smells were more change prone and required more maintenance effort than the classes with no code smells. Deligiannis et al. [25] investigated the god class code smells on the design of object-oriented systems by 12 students in a classroom setting. They reported that good design without code smell helped to make the design more understandable and maintainable. Khomh et al. [47] used DECOR [60] to detect 29 types of code smells and investigate their relationships with change-proneness. The study was conducted on the change history of two open source systems Azureus and Eclipse. They

concluded that, in general, classes with smells are more prone to changes than classes without smells on the 9 versions of Azureus and 13 versions of Eclipse.

Li et al. [53] studied Eclipse to determine the relationship between code smells and different severity levels of defects. Their results indicate that the large class, large method, and shotgun surgery code smell show significant association with all severity levels of software defects and that data class, refused bequest, and feature envy code smells are not associated significantly with software defects or particular severity levels of software defects. Therefore, not all types of code smell lead to defects. Olbrich et al. [68] reported that god classes and brain classes code smells of open source systems Lucene, Xerces, and Log4j were less prone to changes and have lesser defects than other classes when normalized with respect to size. Therefore, for reasonably-sized god and brain classes, the code smells should not be problematic. Similarly, Sjøberg [80] did not find any significant increase in maintenance effort with files having code smells after controlling file size and number of changes. In fact, refused bequest experienced decreased maintenance effort.

5.2.4 Antipatterns

A design pattern becomes an antipattern when it causes more problems than it solves. Essentially, antipatterns indicate design weaknesses that could potentially increase the risk of faults in the software later and make systems harder to maintain. Antipatterns identify poor design solutions at a higher level of abstraction than code smells. Some antipatterns are the blob, spaghetti code, lava flow, dead end, cut-and-paste programming, and mush-

room management [8]. DÉCOR (Detection and CORrection) is a method proposed by Moha et al. [60] to detect antipatterns.

We provide an overview of a few studies on antipatterns and their effect on code maintainability. Romano et al. [72] studied 16 Java open source systems to determine if classes with antipatterns are more change prone than classes with no antipattern. They reported that classes with antipatterns do in fact change more frequently. They also found that certain type of antipatterns such as ComplexClass, SpaghettiCode, and SwissArmyKnife are more change prone than other types of antipatterns. Abbes et al. [1] investigated the effect of the Blob and SpaghettiCode antipatterns on program comprehension based on different Java systems. Each antipattern on its own was not very harmful to program comprehension, but the combination of both had a negative impact on program comprehension and, subsequently, can result in the introduction of defects. Khomh et al. [48] studied the relationship between 13 antipatterns and change- and defect-proneness in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino. They found that classes with antipatterns are more change- and defect prone than other classes. They also reported that the Blob, ComplexClass, and LargeClass are correlated with one another. A similar study by Jaafar et al. [43] was conducted to evaluate whether classes with antipatterns are more defect-prone using open source systems ArgoUML, JFreeChart, and XercesJ. They concluded that antipatterns resulted in more defect prone classes. Taba et al. [84] studied how antipatterns can be used for defect prediction using Eclipse and ArgoUML systems. They confirmed that antipattern classes are more defect prone than other classes and proposed 4 metrics that can be used to extract antipatterns from source code. They concluded that the antipattern

metrics provide additional information about defect proneness of a system in addition to traditional metrics.

5.2.5 Metrics

All of these four indicators discussed here are metrics (both traditional and Object Oriented) expressed at a certain threshold. Several studies used metrics, both traditional and Object Oriented metrics, as quality indicators, especially to indicate change- and defect-proneness. Defect and fault proneness are used interchangeably in this dissertation. Next, we survey the literature to understand how metrics are used to indicate problems in the source code.

Basili et al. [5] assessed whether the CK metrics could predict fault-proneness using 8 medium-sized systems. They used logistic regression, a standard classification technique based on maximum likelihood estimation to assess the relationship between defects and fault-proneness. They concluded that 5 out of the CK metrics, namely WMC, DIT, RFC, NOC, and CBO, were correlated with defect-proneness. Lack of Cohesion of Methods (LCOM) was insignificant. Briand et al. [6] investigated the relationship between coupling, cohesion, and inheritance metrics and fault-proneness in an industrial system using univariate logistic regression. They determined that the metrics measures, coupling, inheritance, and size were associated with fault-proneness, but not cohesion. Catal et al. [11] examined the impact of CK metrics and some method-level metrics on fault-proneness using the Artificial Immune Recognition System (AIRS). They concluded that the combination of CK and the lines of code metrics provide the best prediction results for fault-proneness.

Pai et al. [70] uses the CK metrics and class level size metric to determine fault-proneness using a Bayesian Network model. They reported that CBO, RFC, WMC, and SLOC were indicators of fault-proneness in the system. Singh et al. [79] performed a similar study but used decision trees as their evaluation model instead. They reported that CBO, RFC, LCOM, WMC, and SLOC were correlated to fault proneness while NOC and DIT were not significant. Cartwright et al. [10] investigated the relationship between classes in inheritance relationships and defect proneness in a C++ system. They found out that classes with inheritance structures are three times more prone to defect than classes with no inheritance structures. They build their prediction models relating size and defects using linear regression. Gyimothy et al. [34] used the CK metrics to predict the defect-proneness of Mozilla using logistic regression and machine learning techniques such as artificial neural networks and decision trees. They found that CBO and LOC were important variables for predicting defects. Zimmermann et al. [94] conducted experiments on Eclipse to determine the relationship between complexity metrics and defect proneness using linear regression models. They concluded that complexity can be used to predict defects and that the more complex the code is, the more defects it has.

Table 5.1 summarizes these studies in terms of the independent variable and predictive models used.

There are many other studies with similar results as reported in the literature review by Jabangwe et al. [44]. They pointed out that 80% of the studies used CK metrics to evaluate fault-proneness in software systems. The most commonly used prediction model was some

Table 5.1

Studies Investigating Metrics and Defect-Proneness

Study	**Independent Variable**	**Predictive Model**
Basili et al. [5]	CK Metrics	Logistic Regression
Briand et al. [6]	Coupling and Cohesion and Inheritance Metrics	Univariate Logistic Regression
Catal et al. [11]	CK Metrics and LOC	Artificial Immune Recognition System (AIRS)
Pai et al. [70]	CK Metrics and SLOC	Bayesian Network Model
Singh et al. [79]	CK Metrics and SLOC	Decision Trees
Cartwright et al. [10]	Inheritance	Linear Regression
Gyimothy et al. [34]	CK Metrics and SLOC	Logistic Regression and Machine Learning
Zimmermann et al. [94]	Complexity Metrics	Linear Regression

form of regression analysis (multivariate logistic or binomial logistics). Very few studies used machine learning models such as decision trees, random forests, or Bayes networks.

Regarding technical debt indicators, modularity violations, and grime are the two most under-studied problems. To our knowledge, there are a few empirical studies on grime but no tool to measure it. Similarly, there is only one study on modularity violations that proposes a tool (CLIO) to pinpoint such problems. For antipatterns, there are a few studies that report strong correlation with defects, and DÉCOR is the only tool available to extract anti-patterns.

The technical debt indicator that we use is a combination of metrics: measure of code smells, defect count, change count, and object-oriented metrics. Code smells are the most widely studied technical debt indicator and is very useful in indicating deeper problems in the source code. A quick search on IEEExplore came up with 104 research papers on code smells since 2012. Despite the diverging opinions on code smell not being the most accurate representation of technical debt at the code level, it is the one most extensively used in industry due to the fact that it is easily obtained. There exist multiple tools that are readily available to easily extract code smells from the source code.

We want this framework to be as simple and easy to use as possible. Therefore, with code smells extraction tools readily available, there is no additional effort required from the project managers for building custom tools, and no expertise is required to extract the code smell data unlike analyzing architecture blueprints to identify technical problems and potential sources of technical debt.

However, not all code smells are representative of problems in source code as some of the studies report. Therefore, we use change count, defect count, and object-oriented metrics, which have been extensively used to indicate problems in source code as a further refinement to get a closer indicator of technical debt.

5.3 RQ3: How do we build a framework to determine the most critical technical debt items?

We will provide insights for this research question based on our historical data extraction described in Chapter 4. We will describe how we put together the framework for both Apache Hive and Apache Mahout.

5.3.1 Apache Hive

For Apache Hive, the data extraction step resulted in 12402 unique classes with their associated variables including change count, defect count, and technical debt items count. Table 5.2 display the overall descriptive statistics of Apache Hive.

After cleaning out the final dataset for anomalies (e.g. classes with no defect data due to missing revision number in the issue tracking system), 9852 classes were left. Out of these 9852 classes, 817 (about 8.3%) had one or more instances of technical debt (TDIs).

Based on the information from the final dataset, we computed the node probability table (NPTs) for the Bayesian Network. Table 5.3 to Table 5.8 presents the NPTs for the different variables. For example, for the CBO variable, there is a 5.8% probability that the class has no technical debt if the CBO value is within the recommended threshold. If the

CBO value is not within the recommended thresholds, there is an 82.1% chance that the class has technical debt .

Table 5.2

Univariate Descriptive Statistics: Apache Hive

Variable	Mean	Std. Deviation	Minimum	Maximum
CBO	5.05	10.15	0	344
LOC	112.21	1354.97	0	130551
LCOM	28.67	36.49	0	100
WMC	15.95	38.59	0	1634
Change Count	1.32	8.04	0	402
Defect Count	1.27	7.81	0	390
TDI Count	0.16	1.22	0	79

N = 9852

Table 5.3

NPT: Apache Hive (CBO)

	No Technical Debt	**Technical Debt**
Recommended	0.058	0.179
Not Recommended	0.942	0.821

Table 5.4

NPT: Apache Hive (LOC)

	No Technical Debt	**Technical Debt**
Recommended	0.095	0.461
Not Recommended	0.905	0.539

Table 5.5

NPT: Apache Hive (LCOM)

	No Technical Debt	**Technical Debt**
Recommended	0.387	0.741
Not Recommended	0.613	0.259

Table 5.6

NPT: Apache Hive (WMC)

	No Technical Debt	**Technical Debt**
Recommended	0.194	0.684
Not Recommended	0.806	0.316

Table 5.7

NPT: Apache Hive (Defect Count)

	No Technical Debt	**Technical Debt**
Recommended	0.132	0.282
Not Recommended	0.868	0.718

Table 5.8

NPT: Apache Hive (Change Count)

	No Technical Debt	**Technical Debt**
Recommended	0.133	0.282
Not Recommended	0.867	0.718

5.3.1.1 Bayesian Network

Prior to building the BN, we had identified our variables of interest using stepwise regression. For Apache Hive, the relevant metrics were CBO, LOC, LCOM, WMC, Change Count, Defect Count and TDI Count metrics (see Table 2.2 for object-oriented metrics definitions). After building the BN, we confirmed this set of relevant variables by performing a sensitivity analysis. Therefore, for the identified versions, the final dataset contains the following variables:

- List of unique classes
- LOC
- WMC
- CBO
- LCOM
- Change count
- Defect count
- TD instances count

Next, we determined what values they can take as illustrated in Table 5.9.

The next step is to determine the structure of the network to capture qualitative relationships between the variables. In particular, two nodes should be connected directly if one affects or causes the other, with the arc indicating the direction of the effect. So, in the BN

Table 5.9

Apache Hive - Bayesian Network Nodes' Values

Node Name	Type	Values
LOC	Binary	{recommended, non-recommended}
WMC	Binary	{recommended, non-recommended}
CBO	Binary	{recommended, non-recommended}
LCOM	Binary	{recommended, non-recommended}
Change Count	Binary	{recommended, non-recommended}
Defect Count	Binary	{recommended, non-recommended}
Technical Debt	Boolean	{T, F}

for Apache Hive (Figure 5.1), we might ask what factors affects a class's chance of having technical debt? The answer would be LOC, WMC, CBO, LCOM, Change Count and Defect Count. Therefore, we have arcs from LOC, WMC, CBO, LCOM, Change Count and Defect Count to Technical Debt. Once we have specified the topology of the BN, we need to quantify the relationships between the different nodes. Thus, we specify the NPT for each node. The root node (technical debt) also has an associated NPT, representing its prior probability. After building the BN, we simulate different scenarios to observe the probability of technical debt being true for the 817 different classes. With 6 variables, we generated a total of 64 different scenarios and their respective probability of fulfilling the goal.

An example scenario as depicted in Figure 5.2 shows CBO, LOC, and WMC being within the recommended thresholds and LCOM, defect count and change count not being within the recommended thresholds. The probability that technical debt being true is 12.7%.

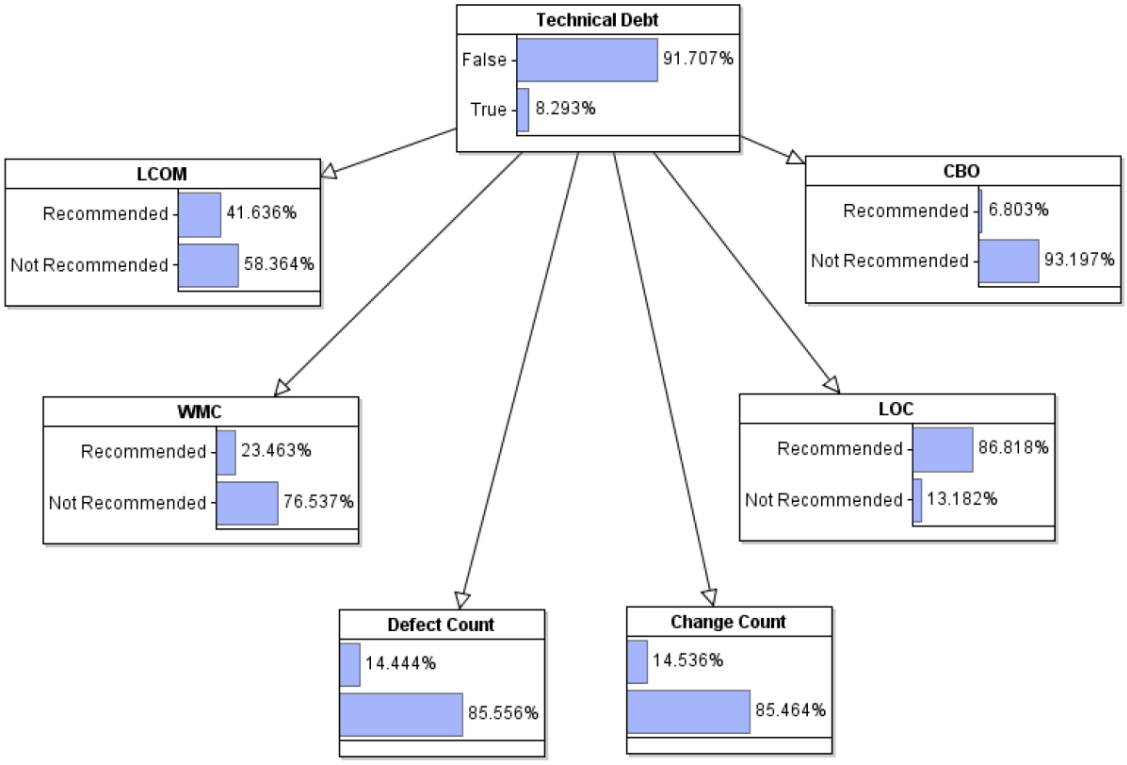

Figure 5.1

Bayesian Network - Apache Hive

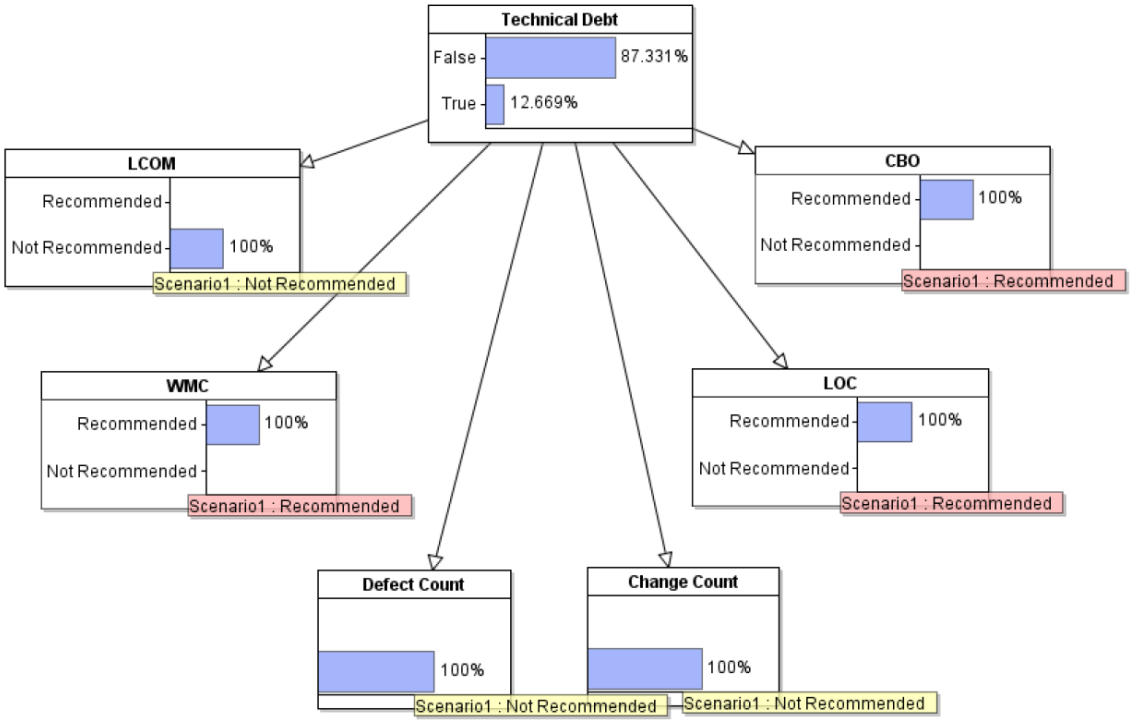

Figure 5.2

Bayesian Network - Apache Hive (Sample Scienario)

5.3.1.2 Classification Scheme

Once we obtained 817 TDIs and their respective probability of technical debt being true, we devised our classification scheme. The range of probability is 0.01 to 0.98. As a result, we devised the classification scheme based on this information:

- Low risk: Probability between 0.0 (inclusive) and 0.4 (exclusive)
- Medium risk: Probability between 0.4 (inclusive) and 0.8 (exclusive)
- High risk: Probability between 0.8 and 1.0 (inclusive)

Table 5.10 depicts the number of TDI according to the different categories of the classification scheme.

Table 5.10

Classification Scheme for Apache Hive

Classification	TDI Count
Low risk	512
Medium risk	176
High risk	129

5.3.1.3 AHP

Prior to applying AHP, we further reduce the dataset to about 5 TDI. In this case, we reduced the debt items based on the interdependence of the debt item with respect to other classes. The resulting dataset is shown in Table 5.11.

To start with, we finalized the following prioritization criteria for the AHP algorithm. We ignored the debt probability as it is the same for all 5 TDIs.

- Impact of the debt (CBO)
- Code churn (Change Count)
- Defect count
- Technical Debt Instances Count

Next, we structure the problem as a hierarchy. In the first (top) level is the goal of most critical TDI. In the second level are the four prioritization criteria which contribute to the goal, and the third (bottom) level are the five candidates which are to be evaluated in terms of the criteria in the second level. The hierarchy is shown in Figure 5.3.

The next step is to perform a pairwise comparison to determine the relative importance of each candidate in terms of each criterion. For instance, the decision maker must ask questions to decide if TD13 is more important than TD52 in terms of impact. Then we will compare the criteria with respect to their importance in reaching the goal.

Pairwise comparisons are quantified using a scale of numbers that indicates how many times more important one element is over another element with respect to the criterion to

Table 5.11

AHP Dataset - Apache Hive

ID	Name	CBO	Defect Count	Change Count	TDI Count	Probability
13	org.apache.hadoop.hive.metastore.HiveMetaStoreClient	111	82	83	6	0.81
17	org.apache.hadoop.hive.metastore.ObjectStore	107	111	115	51	0.81
23	org.apache.hadoop.hive.ql.Driver	103	141	141	1	0.81
52	org.apache.hadoop.hive.metadata.Hive	86	115	117	6	0.81
65	org.apache.hadoop.hive.ql.parse.SemanticAnalyzer	236	333	345	13	0.81

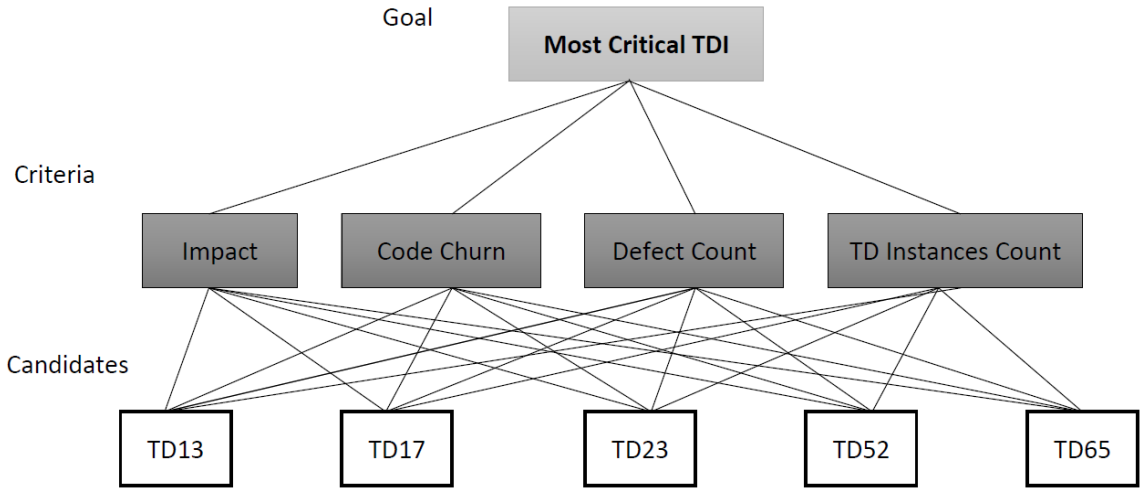

Figure 5.3

Problem Hierarchy - Apache Hive

which they are compared. For instance, if TD13 is moderately more important than TD52 in terms of impact, a value of 3 is entered in its appropriate position and the reciprocal, $\frac{1}{3}$, is entered in the transpose position. Table 5.12 depicts the scale proposed by Saaty [73].

Table 5.14 to Table 5.17 depicts the pairwise comparison matrices of the five candidates: TD13, TD17, TD23, TD52, and TD65, in terms of the four criteria: impact, code churn, defect count and technical debt instances count, along with the priority vector.

The weights of importance of the four criteria with respect to the overall goal are also determined by using pairwise comparisons. The pairwise comparison matrix for the four criteria along with the priority vector is presented in Table 5.18.

The vector of priorities is the principal eigenvector of the matrix. It gives the relative priority of the criteria measured on a ratio scale. In addition, AHP checks for the consistency of the pairwise comparisons using the consistency ratio (CR). If CR is less than 0.1

Table 5.12

Scale of Relative Importances

Intensity of importance	**Definition**
1	Equal importance
3	Moderate importance
5	Strong importance
7	Very strong importance
9	Extreme importance
2,4,6,8	Intermediate values

(10%), then the pairwise comparisons are considered to be adequately consistent. The CR calculation is based on the consistency index (CI). CI is calculated as follows: adding the columns in the judgment matrix, multiply the resulting vector by the vector of priorities, resulting in λ_{max}. CI is then calculated using the formula [73]:

$$CI = \frac{\lambda max - n}{n - 1} \tag{5.1}$$

Next, CR is obtained by dividing CI by the Random Consistency Index given in Table 5.13.

Now that we know the priorities of the criteria with respect to the goal, and the priorities of the candidates with respect to the criteria, we can calculate the priorities of the candidates with respect to the goal. We achieve that for each candidate by adding

- the candidate's priority with respect to impact multiplied by impact's priority with respect to the goal
- the candidate's priority with respect to change count multiplied by change count's priority with respect to the goal
- the candidate's priority with respect to defect count multiplied by defect count's priority with respect to the goal
- the candidate's priority with respect to technical debt instances count multiplied by technical debt instances count's priority with respect to the goal

The overall priorities for all the candidates are depicted in Table 5.19.

Based on the overall priorities table, TD65 has the highest probability of being critical, followed by TD17, TD23, TD13, and TD52. Therefore, this is the order in which these technical debt items for Apache Hive should be tackled.

Table 5.13

Random Consistency Index

n	RCI
1	0
2	0
3	0.58
4	0.90
5	1.12
6	1.24
7	1.32
8	1.41
9	1.45

Table 5.14

Apache Hive Pairwise Comparison Based on Criteria: Impact

	TD13	**TD17**	**TD23**	**TD52**	**TD65**	**Priority Vector**
TD13	1.00	1.00	1.00	3.00	0.14	0.10
TD17	1.00	1.00	1.00	3.00	0.11	0.10
TD23	1.00	1.00	1.00	3.00	0.11	0.10
TD52	0.33	0.33	0.33	1.00	0.11	0.04
TD65	7.00	9.00	9.00	9.00	1.00	0.66

CR = 0.030

Table 5.15

Apache Hive Pairwise Comparison Based on Criteria: Change Count

	TD13	**TD17**	**TD23**	**TD52**	**TD65**	**Priority Vector**
TD13	1.00	0.33	0.20	0.33	0.11	0.04
TD17	3.00	1.00	0.33	1.00	0.11	0.08
TD23	5.00	3.00	1.00	3.00	0.11	0.17
TD52	3.00	1.00	0.33	1.00	0.11	0.08
TD65	9.00	9.00	9.00	9.00	1.00	0.64

CR = 0.077

Table 5.16

Apache Hive Pairwise Comparison Based on Criteria: Defect Count

	TD13	**TD17**	**TD23**	**TD52**	**TD65**	**Priority Vector**
TD13	1.00	0.33	0.20	0.33	0.11	0.04
TD17	3.00	1.00	0.33	1.00	0.11	0.08
TD23	5.00	3.00	1.00	3.00	0.11	0.17
TD52	3.00	1.00	0.33	1.00	0.11	0.08
TD65	9.00	9.00	9.00	9.00	1.00	0.64

CR = 0.077

Table 5.17

Apache Hive Pairwise Comparison Based on Criteria: Technical Debt Instances Count

	TD13	**TD17**	**TD23**	**TD52**	**TD65**	**Priority Vector**
TD13	1.00	0.11	3.00	1.00	0.33	0.08
TD17	9.00	1.00	9.00	9.00	7.00	0.64
TD23	0.33	0.11	1.00	0.33	0.20	0.04
TD52	1.00	0.11	1.00	1.00	0.33	0.06
TD65	3.00	0.14	5.00	3.00	1.00	0.18

CR = 0.016

Table 5.18

Apache Hive Pairwise Comparison for Criteria

	Impact	**Change Count**	**Defect Count**	**Technical Debt Intances Count**	**Priority Vector**
Impact	1.00	5.00	5.00	3.00	0.55
Code Churn	0.20	1.00	1.00	0.33	0.10
Defect Count	0.20	1.00	1.00	0.33	0.10
Technical Debt Instances Count	0.33	3.00	3.00	1.00	0.25

CR = 0.016

Table 5.19

Apache Hive Overall Priorities

	Impact	**Change Count**	**Defect Count**	**Technical Debt Intances Count**	**Goal**
TD13	0.057	0.004	0.004	0.021	0.085
TD17	0.055	0.008	0.008	0.161	0.231
TD23	0.055	0.016	0.016	0.010	0.096
TD52	0.024	0.008	0.008	0.015	0.054
TD65	0.365	0.062	0.062	0.045	0.534

5.3.2 Apache Mahout

For Apache Mahout, the data extraction step resulted in 2794 unique classes with their associated variables, including change count, defect count, and technical debt items count. Table 5.20 display the overall descriptive statistics of Apache Mahout.

After cleaning out the final dataset for anomalies, 1920 classes were left. Out of these 1920 classes, 186 (about 9.7%) had one or more instances of technical debt (TDIs).

Based on the information from the final dataset, we computed the NPTs for the Bayesian Network. Table 5.21 to Table 5.26 present the NPTs for the different variables. For example, for the LOC variable, there is a 4.3% probability that the class has no technical debt if the LOC value is within the recommended threshold, and 78% that the class has technical debt if the LOC value is not within the recommended thresholds.

Table 5.20

Univariate Descriptive Statistics: Apache Mahout

Variable	Mean	Std. Deviation	Minimum	Maximum
NOC	0.52	4.14	0	125
LOC	60.44	92.71	0	1698
LCOM	23.92	30.96	0	100
WMC	9.83	16.32	0	385
Change Count	2.19	5.00	0	62
Defect Count	1.33	2.94	0	30
TDI Count	0.23	1.13	0	18

N = 1920

Table 5.21

NPT: Apache Mahout (NOC)

	No Technical Debt	Technical Debt
Recommended	0.045	0.027
Not Recommended	0.955	0.973

Table 5.22

NPT: Apache Mahout (LOC)

	No Technical Debt	**Technical Debt**
Recommended	0.043	0.220
Not Recommended	0.957	0.780

Table 5.23

NPT: Apache Mahout (LCOM)

	No Technical Debt	**Technical Debt**
Recommended	0.377	0.683
Not Recommended	0.623	0.317

Table 5.24

NPT: Apache Mahout (WMC)

	No Technical Debt	**Technical Debt**
Recommended	0.140	0.409
Not Recommended	0.860	0.591

Table 5.25

NPT: Apache Mahout (Defect Count)

	No Technical Debt	**Technical Debt**
Recommended	0.288	0.565
Not Recommended	0.712	0.435

Table 5.26

NPT: Apache Mahout (Change Count)

	No Technical Debt	**Technical Debt**
Recommended	0.223	0.462
Not Recommended	0.777	0.538

5.3.2.1 Bayesian Network

Prior to building the BN, we had identified our variables of interest using stepwise regression. For Apache Mahout, the relevant metrics were NOC, LOC, LCOM, WMC, Change Count, Defect Count and TDI Count metrics. After building the BN, we confirmed this set of relevant variables by performing a sensitivity analysis. Therefore, for the identified versions, the final dataset contains the following variables:

- List of unique classes
- LOC
- WMC
- NOC
- LCOM
- Change count
- Defect count
- TD instances count

We then determined what values they can take as illustrated in Table 5.27.

The next step is to determine the structure of the network to capture qualitative relationships between the variables. In particular, two nodes should be connected directly if one affects or causes the other, with the arc indicating the direction of the effect. So, in the BN

Table 5.27

Apache Mahout - Bayesian Network Nodes' Values

Node Name	Type	Values
LOC	Binary	{recommended, non-recommended}
WMC	Binary	{recommended, non-recommended}
NOC	Binary	{recommended, non-recommended}
LCOM	Binary	{recommended, non-recommended}
Change Count	Binary	{recommended, non-recommended}
Defect Count	Binary	{recommended, non-recommended}
Technical Debt	Boolean	{T, F}

for Apache Mahout (Figure 5.4), we might ask what factors affects a class's chance of having technical debt? The answer would be LOC, WMC, NOC, LCOM, Change Count and Defect Count. Therefore, we have arcs from LOC, WMC, NOC, LCOM, Change Count and Defect Count to Technical Debt. Once we have specified the topology of the BN, we need to quantify the relationships between the different nodes. Thus, we specify the NPT for each node. The root node (technical debt) also has an associated NPT, representing its prior probability. After building the BN, we simulate different scenarios to observe the probability of technical debt being true for the 186 different classes. With 6 variables, we generated a total of 64 different scenarios and their respective probability of fulfilling the goal.

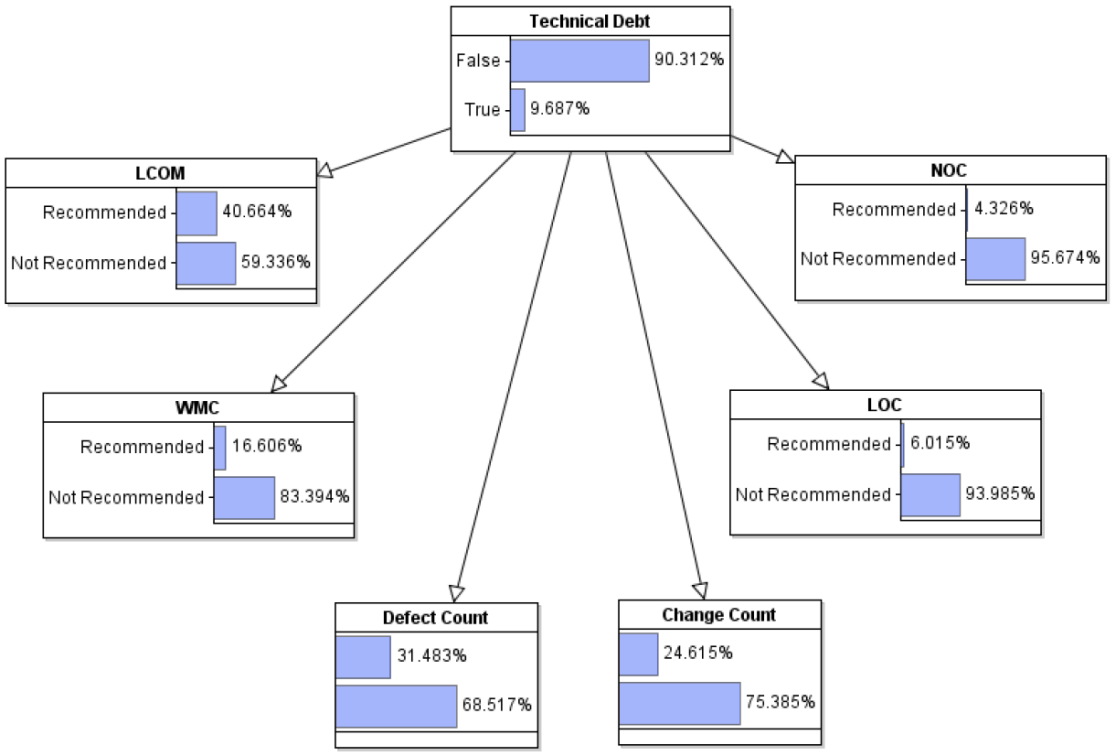

Figure 5.4

Bayesian Network - Apache Mahout

An example scenario as depicted in Figure 5.5 shows change count and WMC being within the recommended thresholds and LCOM, LOC, defect count and NOC not being within the recommended thresholds. The probability that of technical debt being true is 14.4%.

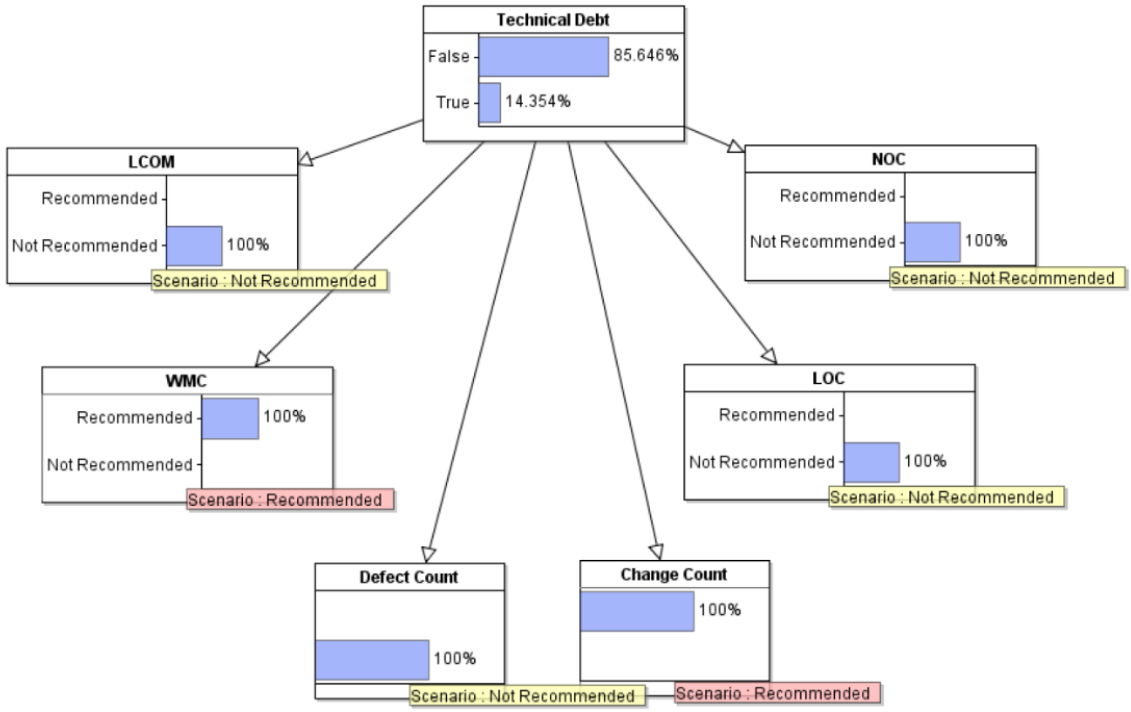

Figure 5.5

Bayesian Network - Apache Mahout (Sample Scenario)

5.3.2.2 Classification Scheme

Once we obtained 186 TDIs and their respective probability of technical debt being true, we devised our classification scheme. The range of probability is 0.01 to 0.92. As a result, we devised the classification scheme based on this information:

- Low risk: Probability between 0.0 (inclusive) and 0.4 (exclusive)
- Medium risk: Probability between 0.4 (inclusive) and 0.7 (exclusive)
- High risk: Probability between 0.7 and 1.0 (inclusive)

Table 5.28 depicts the number of TDI according to the different categories of the classification scheme.

Table 5.28

Classification Scheme for Apache Hive

Classification	TDI Count
Low risk	135
Medium risk	27
High risk	25

5.3.2.3 AHP

Prior to applying AHP, we chose the top 5 debt items with the highest defect count. The resulting dataset is shown in Table 5.29.

Table 5.29

AHP Dataset - Apache Mahout

ID	Name	LOC	Defect Count	Change Count	Probability
20	org.apache.mahout.common.AbstractJob	320	27	42	87.63
83	org.apache.mahout.clustering.canopy.CanopyDriver	179	22	51	65.71
92	org.apache.mahout.clustering.fuzzykmeans.FuzzyKMeansDriver	172	29	59	65.71
96	org.apache.mahout.clustering.kmeans.KMeansDriver	115	30	62	65.71
100	org.apache.mahout.clustering.kmeans.TestKmeansClustering	243	22	39	92.32

To start with, we finalized the following prioritization criteria for the AHP algorithm.

- Size of class (LOC)
- Code churn (Change Count)
- Defect count
- Probability (of technical debt being true)

Next, we structure the problem as a hierarchy. In the first (top) level is the goal of most critical TDI. In the second level are the four prioritization criteria which contribute to the goal, and the third (bottom) level are the five candidates which are to be evaluated in terms of the criteria in the second level. The hierarchy is shown in Figure 5.6.

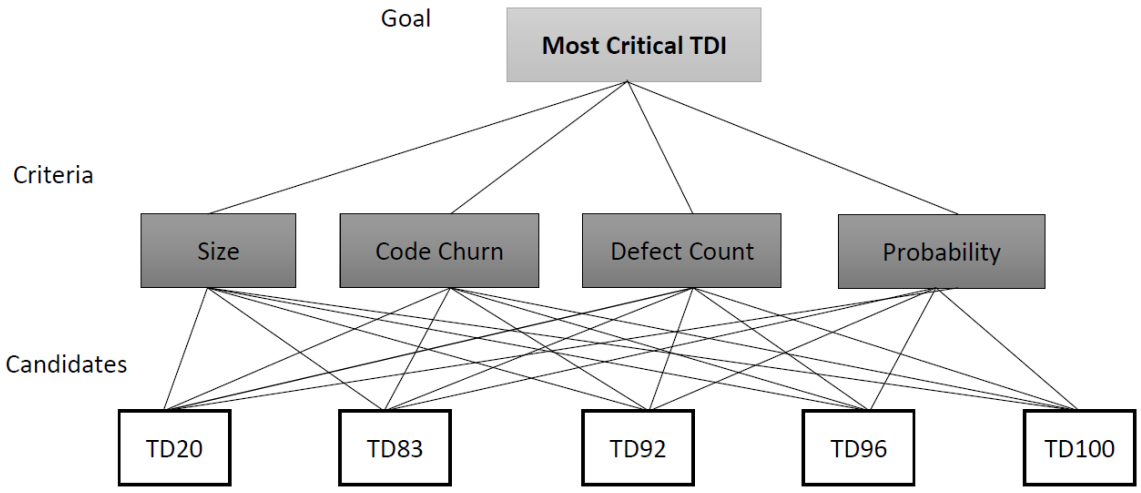

Figure 5.6

Problem Hierarchy - Apache Mahout

The next step is to perform a pairwise comparison to determine the relative importance of each candidate in terms of each criterion. For instance, the decision maker may ask questions decide if TD83 is more problematic than TD96 in terms of size. Then, we will compare the criteria with respect to their importance in reaching the goal.

Pairwise comparisons are quantified using a scale of numbers that indicates how many times more important one element is over another element with respect to the criterion to which they are compared. For instance, if TD83 is more problematic than TD96 in terms of size, a value of 3 is entered in its appropriate position and the reciprocal, $\frac{1}{3}$, is entered in the transpose position. We use Table 5.12 from the previous section for the scale proposed by Saaty [73].

Table 5.30 to Table 5.33 depicts the pairwise comparison matrices of the five candidates: TD20, TD83, TD92, TD96, and TD100, in terms of the four criteria: size, code churn, defect count and probability, along with the priority vector.

The weights of importance of the four criteria with respect to the overall goal are also determined by using pairwise comparisons. So, the pairwise comparison matrix for the four criteria along with the priority vector is presented in Table 5.34. Similar to Apache Hive, we calculated the CR for Apache Mahout. We used the Random Consistency Index given in Table 5.13.

Now that we know the priorities of the criteria with respect to the goal, and the priorities of the candidates with respect to the criteria, we can calculate the priorities of the candidates with respect to the goal. We achieve that for each candidate by adding

- the candidate's priority with respect to size multiplied by size's priority with respect to the goal
- the candidate's priority with respect to change count multiplied by change count's priority with respect to the goal
- the candidate's priority with respect to defect count multiplied by defect count's priority with respect to the goal
- the candidate's priority with respect to probability multiplied by probability's priority with respect to the goal

The overall priorities for all the candidates are depicted in Table 5.35.

Based on the overall priorities table, TD20 has the highest probability of being critical, followed by TD92, TD96, TD100, and TD83. Therefore, this is the order in which these technical debt items for Apache Mahout should be tackled.

The framework depicts the process to be followed from identification to prioritization of the technical debt items. The process is similar irrespective of the software system but the way the framework is applied results in different variables being relevant for different systems.

Table 5.30

Apache Mahout Pairwise Comparison Based on Criteria: Size

	TD20	**TD83**	**TD92**	**TD96**	**TD100**	**Priority Vector**
TD20	1.00	5.00	5.00	7.00	3.00	0.48
TD83	0.20	1.00	1.00	3.00	0.33	0.10
TD92	0.20	1.00	1.00	3.00	0.20	0.10
TD96	0.14	0.33	0.33	1.00	0.20	0.05
TD100	0.33	3.00	5.00	5.00	1.00	0.27

CR = 0.047

Table 5.31

Apache Mahout Pairwise Comparison Based on Criteria: Change Count

	TD20	**TD83**	**TD92**	**TD96**	**TD100**	**Priority Vector**
TD20	1.00	0.33	0.20	0.20	1.00	0.06
TD83	3.00	1.00	0.33	0.33	5.00	0.17
TD92	5.00	3.00	1.00	1.00	5.00	0.35
TD96	5.00	3.00	1.00	1.00	5.00	0.35
TD100	1.00	0.20	0.20	0.20	1.00	0.06

CR = 0.031

Table 5.32

Apache Mahout Pairwise Comparison Based on Criteria: Defect Count

	TD20	**TD83**	**TD92**	**TD96**	**TD100**	**Priority Vector**
TD20	1.00	3.00	1.00	1.00	3.00	0.27
TD83	0.33	1.00	0.33	0.33	1.00	0.09
TD92	1.00	3.00	1.00	1.00	3.00	0.27
TD96	1.00	3.00	1.00	1.00	3.00	0.27
TD100	0.33	1.00	0.33	0.33	1.00	0.09

CR = 0.0

Table 5.33

Apache Mahout Pairwise Comparison Based on Criteria: Probability

	TD20	**TD83**	**TD92**	**TD96**	**TD100**	**Priority Vector**
TD20	1.00	3.00	3.00	3.00	1.00	0.33
TD83	0.33	1.00	1.00	1.00	0.33	0.11
TD92	0.33	1.00	1.00	1.00	0.33	0.11
TD96	0.33	1.00	1.00	1.00	0.33	0.11
TD100	1.00	3.00	3.00	3.00	1.00	0.33

CR = 0.016

Table 5.34

Apache Mahout Pairwise Comparison for Criteria

	Size	**Change Count**	**Defect Count**	**Probability**	**Priority Vector**
Size	1.00	3.00	0.20	0.33	0.12
Change Count	0.33	1.00	0.14	0.20	0.06
Defect Count	5.00	7.00	1.00	3.00	0.56
Probability	3.00	5.00	0.33	1.00	0.26

CR = 0.044

Table 5.35

Apache Mahout Overall Priorities

	Size	**Change Count**	**Defect Count**	**Probability**	**Goal**
TD20	0.059	0.004	0.152	0.088	0.303
TD83	0.012	0.010	0.051	0.029	0.102
TD92	0.012	0.020	0.152	0.029	0.213
TD96	0.006	0.020	0.152	0.029	0.207
TD100	0.033	0.033	0.051	0.029	0.175

5.4 RQ4: How effective is the framework?

Our framework was validated using the static analysis tool, Findbugs. Findbugs is a widely used tool in the industry. For instance, following the Google fixit event in 2009 [4] using Findbugs to flag potential bugs, Findbugs is now part of Google's review process. Findbugs flags over 400 violations in the source code based on different categories. The categories we selected when running Findbugs were bad practice (code that violates good coding practices), malicious code vulnerability (code that can be maliciously altered by other code), correctness (coding mistake that was not what the developer intended), performance (code that could be written better to increase performance), security (code that can cause security problems) and dodgy code (code that leads to errors). In addition, Findbugs assign their bugs a ranking from 1-20, broken down into categories as illustrated in Table 5.36.

We ran Findbugs on the source code of version 2.0.0 of Apache Hive and version 0.12.2 of Apache Mahout respectively. These versions were chosen because they correspond to the versions that the five most critical technical debt items of each system belong to.

When we ran Findbugs on Apache Hive, three of the five technical debt items that our framework flagged as most critical were detected by Findbugs. Findbugs ranked all the classes of Apache Hive as 14 and above.

Similarly, when we ran Findbugs on Apache Mahout, all of the five technical debt items that our framework flagged as most critical were assigned rank 18. In general, all the classes of Apache Mahout were ranked 14 and above. None of the classes was in category scariest or scary for bothe Apache Hive and Mahout.

Table 5.36

Findbugs Categories

Category	Rank
Scariest	1-4
Scary	5-9
Troubling	10-14
Of Concern	15-20

Findbugs flagged 6 classes with rank 14 for Apache Mahout, and we investigated these classes. We extracted their metrics as shown in Table 5.37.

We found that the classes flagged by Findbugs are not problematic based on the metrics. 2 out of 3 classes were below the recommended threshold for LOC. The values for LCOM were relatively high but we have classes with LCOM of 100 that were not flagged by Findbugs. The highest value for WMC of Mahout classes were 385, and the values of the 3 classes flagged by Findbugs were below 65. When compared to other classes, these 3 classes were not among the ones with the highest values for the different metrics.

Out of the 10 most critical technical debt items flagged by our framework, Findbugs extracted 8 as problematic. Our framework and Findbugs, however, examine different indicators. For instance, one line of code can have a security vulnerability such as use of SHA-1, which is a weak hash function as mandated by NIST[3]. This statement will be

[3]NIST's Policy on Hash Functions, http://csrc.nist.gov/groups/ST/hash/policy.html (accessed Sep. 20, 2016)

Table 5.37

Findbugs Classes with Rank 14

Name	CBO	NOC	LOC	LCOM	WMC	Change Count	Defect Count	TD Instances Count
org.apache.mahout.math.list.ObjectArrayList	3	0	185	46	48	0	0	0
org.apache.mahout.math.map.OpenHashMap	4	0	364	76	63	0	0	1
org.apache.mahout.common IntPairWritable	1	0	194	66	24	7	4	2

an immediate red flag in Findbugs[4] but for our framework, if the code is written using good practices (e.g. no shortcuts taken and no code smell present), then it will not be flagged as problematic by our framework. Our framework and tools such as Findbugs are complementary. While our framework can be used to narrow down the most critical items in a software, Findbugs can be used to figure out what the specific problems are and to which category of problems they belong.

In this chapter, we addressed each research question based on our empirical studies, literature reviews and application of our framework on Apache Hive and Apache Mahout. We used the framework to generate the most critical technical debt items and use Findbugs to evaluate our framework. We concluded that our framework and Findbugs are complementary. In the next chapter, we further elaborate on these results.

[4] Bug Patterns, http://find-sec-bugs.github.io/bugs.htm (accessed Sep. 20, 2016)

CHAPTER 6

DISCUSSION

This chapter will discuss the findings from our empirical studies in aggregate as well as discuss the implications of our framework for the software industry. Lastly, we present the limitations of this work.

6.1 Empirical Studies

We provided insights from our empirical studies on multiple facets on technical debt presented in Chapter 5.

With regards to the prevailing confusion in the technical debt community as to the definition and taxonomy of technical debt, our study shows that technical debt encompasses much more than code and design debt (e.g. we found that most respondents see defects as a type of technical debt). Technical debt goes beyond the boundaries of the definitions we listed in Chapter 1. We illustrate our findings for the technical debt taxonomy using Figure 6.1. This could be used as a baseline towards refining a universal definition of the metaphor.

Most practitioners agree that technical debt costs involve more than just the effort to fix the debt item. Additional cost factors like interest amount, liability costs, and potential litigation costs where service level agreements are violated because of unmanageable

debts should also be included. These factors relating to the financial risks of poor quality software should be considered as part of the decision process to manage technical debt. Furthermore, respondents believe that technical debt is an important issue to discuss and evaluating technical debt should be part of the companies' risk assessment process. The majority of companies, however, do not carry out these practices. Risk assessment is an understudied but important issue in the context of technical debt. Risk is another potential factor that needs to be incorporated in the decision process to manage technical debt. Determining the value of technical debt items in future releases can help companies make timely decisions about which debt items to fix first. Another aspect that was uncovered in our case study is that technical debt management is viewed differently in different types of companies. Therefore, we distinguish between the following type of companies. Mature and established companies would be more concerned about technical debt rather than a new company (startup) which is more concerned with getting the product out of the door.

- New Companies (Startup)
- Mature Companies (Beyond startup stage with consistent revenue)
- Large Companies (Well established with ongoing concerns)

Similarly, there are other criteria that our empirical study [19] helped uncover. We provide a range of criteria that project managers can use as part of the decision-making process when managing technical debt in Figure 6.2. The terminologies used in Figure 6.1 and Figure 6.2 are explained in Table 6.1 and Table 6.2 respectively.

When dealing with technical debt, software practitioners can use the technical debt taxonomy tree to distinguish between different types of debt. In so doing, they will help them manage the debt better when they are aware of the type of debt they are dealing with. Similarly, with our proposed management criteria tree, when practitioners make decisions about which technical debt items need to be tackled, they can choose from the set of criteria according to their business objectives. While we tried to incorporate as many criteria as possible based on existing research and our empirical studies, we understand that our list might not be comprehensive (e.g. some criteria might be domain specific or some criteria can be imposed by the customer).

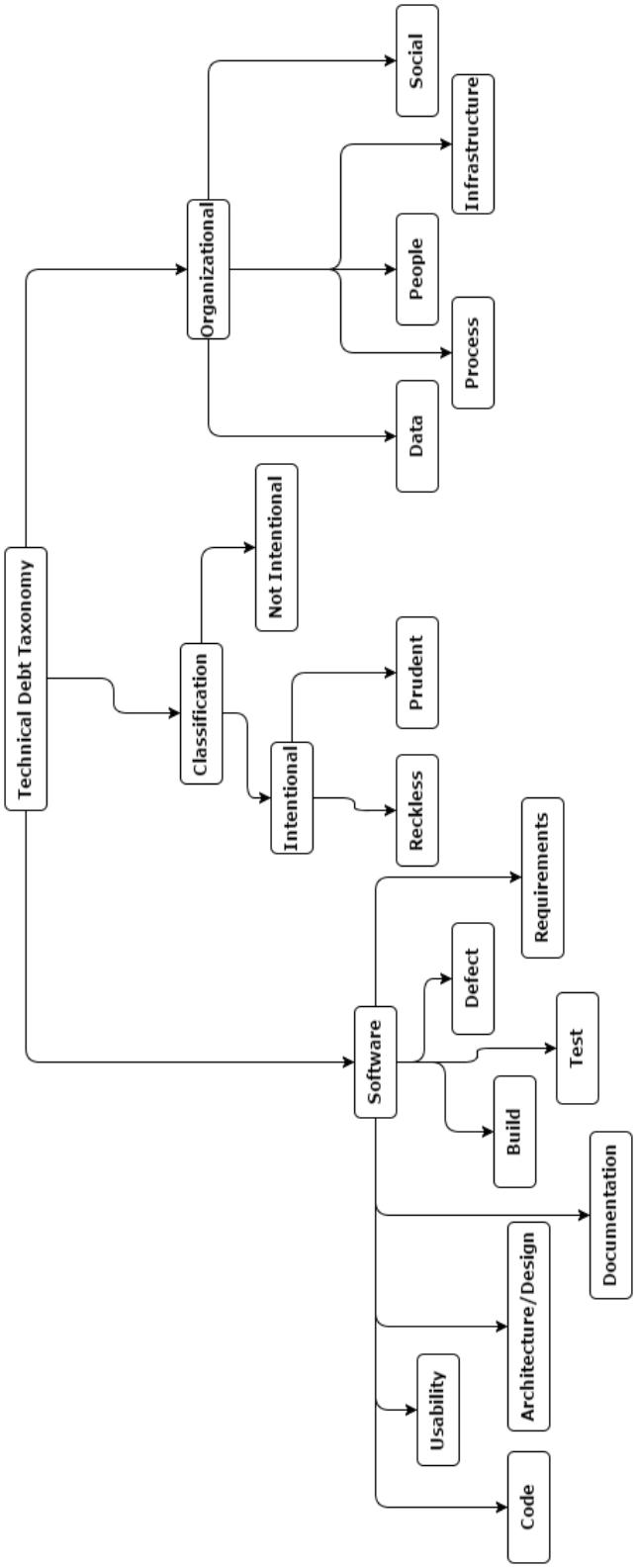

Figure 6.1

Technical Debt Taxonomy

Figure 6.2

Technical Debt Management

Table 6.1

Technical Debt Taxonomy - Descriptions

Types of Technical Debt	**Definition**
Software	Categories of technical debt related to software
Organizational	Categories of technical debt related to the organization
Classification	Categories of technical debt as intentional or non-intentional
Requirements	Tradeoffs made with respect to what requirements the development team need to implement. This category can also include delayed or wrong features.
Architecture/ Design	Refers to software design no longer fits its intended purpose
Code	Refers to violations of design principles in production code
Test	Refers test plan is not completely carried out
Defect	Refers to known defects that are not yet fixed
Build	Refers to flaws in the build system or build process of a software
Usability	Refers to technical debt associated with difficult to user interfaces resulting in inconsistent or poor user experience
Documentation	Refers to missing or inadequate documentation
Data	Refers to technical debt associated with poor quality data
Infrastructure	Refers to delaying an upgrade or infrastructure fix
People	Refers to lack of skills, experience in technology, tools and techniques to build better quality software. Also known as knowledge debt.
Process	Refers to inefficient processes related to the code development and test environment
Social	Refers to the results of sub-optimal socio-technical decisions
Intentional	Refers to technical debt taken on voluntarily
Unintentional	Refers to technical debt taken on involuntarily
Reckless	Refers to debt taken on that has an overall negative impact with time (e.g. increased interest payments)
Prudent	Refers to conscious debt taken on for short term benefits based on a decision that is not sustainable in the long term

Table 6.2

Technical Debt Management - Descriptions

Technical Debt Management Criteria	**Definition**
Liability	Assessing how much impact the technical debt item can potentially have
Business value	Assessing how important the technical debt item (e.g. in terms of the functionality where it is located) is to the business
Performance	Assessing how the debt item affect performance of software (e.g. will the impact happen once, or will it recur?)
Location	Assessing the location of the debt item (e.g. is it located in some critical functionality that is heavily used?)
Scope	Assessing how the technical debt item is relevant to the objective/scope and context of the project
Metrics	Assessing the technical debt item in terms of size, complexity, and dependencies on other classes
Age of debt	Assessing the age of the debt (e.g. is it "legacy TD," or did the team add it during the version?)
Age of company	Assessing the importance of technical debt for the company (e.g. for a mature company, managing technical debt is usually important)
Developer's productivity	Assessing how the debt item affect performance of developers or slow them down
Type of debt	Assessing the type of debt to better manage it (see technical debt taxonomy tree)
Visibility of debt	Assessing how visible the debt item is (e.g. is it located in a feature that is heavily used? will the impact be perceived very soon, or after delivery?)
Age of software	Assessing the age of the software where the debt is located (e.g. teams will not manage technical debt in legacy software which will retire soon)
Criticality of the debt	Assessing how problematic the technical debt item can be for the project

6.2 Framework

The technical debt decision-making framework proposed was created to assist project managers to manage and prioritize technical debt. The framework consists of different phases. In the first phase, the technical debt items are identified. We propose a unique combination of metrics that can be used for this purpose. Once the technical debt items have been identified, the next step is to build the Bayesian Network to determine how problematic each item is. Following that, a classification scheme is born out of the data to categorize the items into low, medium and high severity. These categories are based on the probability of the items being problematic and this information is obtained from the previous phase. The last phase consists of narrowing down the technical debt items to a manageable amount and using AHP to prioritize these items based on criteria from Figure 6.2 that the team deems most suitable for the project.

One of the benefits of our framework is that it can be customized to fit any project. While the company might already have a manual list of technical debt items, our unique combination of metrics extract code smells as an initial set of technical debt items and further narrow down the items to the most defect and change prone ones. In addition, the bayesian network helps to predict the criticality of each of these debt items. Over time with more data, the accuracy of such prediction models increases. It is often the case in companies that technical debt items are addressed based on what the project manager deems right without any analysis as to which item is really problematic. As a result, the lowest severity items keep on accumulating and can cause software decay in the long term. As a remedy, we propose a classification scheme that further divides the technical debt

items into categories. The project manager can choose to address a subset of each category instead of always focusing on the high severity items. Lastly, the decision model allows the team to decide on the most important criteria to prioritize their technical debt items. These criteria will change for different projects as they will depend on the business objectives. This flexibility implies that with different criteria, different technical debt items can be the most critical.

In addition, our aim was to keep the framework simple so that it can be used regularly during a project's life cycle to prioritize technical debt items. For instance, in a project using agile practices, the framework can be used during the hardening sprint (an additional specialized sprint prior to release focused on defect repair, testing and paying back technical debt) to enable better usage of resources (e.g. time).

Compared to static analysis tools with severity indexes such as inCode and Findbugs, our framework is different because it does not extract problematic items only based on certain rules but using a wide range of criteria. These criteria are customized for the projects based on their business objectives and static analysis tool does not distinguish between different projects. At most, the thresholds of the rules can be changed based on different projects in static analyzers.

6.3 Threats to Validity

This section discusses the limitations and threats to validity in this study.

6.3.1 Construct Validity

Construct validity threats concern the relation between theory and observation. In this dissertation, they are mainly due to errors introduced in the data collection and the empirical studies. To cater for threats to construct validity in our survey, we provided clear definitions of the various terms related to technical debt. In addition, we carefully worded the questions in an unbiased manner, but we acknowledge that there are different ways of describing similar concepts (e.g. participants could have been mistaken with questions referring to measuring technical debt versus tracking technical debt). Regarding the historical data extraction, the actual number of defects might be higher than reported, but as defect data could not be extracted when information such as the revision number was missing in the issue tracking system. Regarding the thresholds we used for the metrics to build the bayesian network, we understand that they are based on studies in the literature, and they may vary according to the programming language. Similarly, for the code smell detection tool, we are aware that the thresholds and metrics used are based on the subjective understanding on the researcher who developed the tool, but we cater for this threat by further refining our technical debt indicator by incorporating other metrics such as change count and defect count.

6.3.2 External Validity

External Validity refers to the ability to generalize results. For the survey, we made sure that more than 50% of the respondents were not from the same company by asking for company names. For building the framework, both Apache Hive and Apache Mahout

are from the Apache Foundation projects and written in Java. Thus, generalization will require further empirical studies. However, we limited such a threat by choosing two systems belonging to different domains of different sizes. Regarding the evaluation of the framework, we recognize the need to further evaluate our framework in an industrial setting.

6.3.3 Internal Validity

This threat refers to the possibility of having unwanted or unanticipated relationships. Two threats to internal validity are present in the survey. The first is selection bias due to the fact that about 25% of the participants were from the same company. This could introduce a confounding factor due to the organization culture and similar workplace practices. The second is researcher bias as the coding process depends on the researcher's interpretation of the interview responses, and this can be influenced by the researcher's experience. To mitigate this bias, the researcher's academic advisor reviewed the results of the coding process prior to the development of the final themes.

CHAPTER 7

CONCLUSIONS

We proposed a decision-making framework that can be used to prioritize technical debt items. We build our framework using empirical data from case studies conducted with a total of 110 software practitioners in industrial settings, literature reviews, and historical data from real open-source systems. The framework outputs the most critical debt items using criteria finalized by the team according to business objectives. The framework can be used in complement to static analysis tools for technical debt management in companies.

7.1 Contributions

This research has the following contributions:

7.1.1 Decision-Making Framework

The framework is one of the major contributions of this dissertation. It is comprehensive, including processes from technical debt identification to technical debt prioritization. It is flexible and customizable based on business objectives. In addition, it allows for teams to reason about risks and technical liability (future impact of the debt).

7.1.2 Prediction Model

The prediction model can be used independently to focus quality assurance effort or assess the criticality of the technical debt item.

7.1.3 Technical Debt Taxonomy

The technical debt taxonomy tree generated based on the findings of our empirical studies proposes the classification of the different types of technical debt based on two broad categories: software and organizational technical debt. Practitioners can use this taxonomy to differentiate between different types of technical debt in their software. Management of technical debt becomes easier when the type of technical debt has been identified.

The technical debt management criteria tree based on the findings of our empirical studies proposes a range of criteria that project managers can use as part of the decision-making process when managing technical debt. These criteria are project and context dependent, and this type of flexibility allows the framework to be applied to different software projects.

7.2 Publications

In this section, we list all the publications related to this dissertation. Some of the works have already been published and are referenced.

7.2.1 Refereed Journal Articles

"Technical Debt Indicators - A Systematic Literature Review" - To be submitted to the Information and Software Technology Journal in Dec 2016

"Technical Debt Decision-Making Framework" - To be submitted to the Empirical Software Engineering Journal in Dec 2016

Zadia Codabux, Byron J. Williams, Murray Cantor, Gary Bradshaw, *"An Empirical Assessment of Technical Debt Practices in Industry,"* Journal of Software: Evolution and Process, 2016 (Under review)

Ajay K. Deo, Zadia Codabux, Kazizakia Sultana, Byron J. Williams, *"Assessing Software Defects Using Nano-Patterns Detection,"* International Journal of Computers and Their Applications (Special Issue on Software Engineering using Data Engineering Approaches), 2016

7.2.2 Refereed Conference Papers

"Technical Debt Decision-Making Framework - An Industrial Evaluation" - To be submitted to ESEM 2017

Zadia Codabux, Byron J. Williams, *"Technical Debt Prioritization using Predictive Analytics,"* 38th International Conference on Software Engineering (ICSE), Companion Volume, Austin, Texas, USA, May 2016

Zadia Codabux, Byron J. Williams, Nan Niu, *"A Quality Assurance Approach to Technical Debt,"* Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, USA, 2014

Isaac Griffith, Derek Reimanis, Clemente Izurieta, Zadia Codabux, Ajay Deo, Byron Williams, *"The Correspondence between Software Quality Models and Technical Debt*

Estimation Approaches," Proceedings of the 6th Workshop on Managing Technical Debt, Victoria, Canada, 2014

Zadia Codabux, *"Technical Debt Decision-Making Framework,"* IEEE ACM IDoESE, 12th International Doctoral Symposium on Empirical Software Engineering, Torino, Italy, 2014

Zadia Codabux, Byron J. Williams, *"Managing technical debt: An Industrial Case Study,"* Proceedings of the 4th Workshop on Managing Technical Debt, San Francisco, USA, 2013, pp. 8-15

7.2.3 Technical Reports

Zadia Codabux, Byron J. Williams, *"Agile Adoption - An Industrial Case Study,"* Technical Report MSU-TR120731, Department of Computer Science and Engineering, Mississippi State University, 2012

7.3 Future Work

This section summarizes how we plan to extend the work presented in this dissertation.

7.3.1 Framework Evaluation

Tools such as inCode and SonarQube provide severity indexes. We plan to use tools other than Findbugs to further compare the results of our framework. This will help us better understand our results and improve the accuracy of our framework.

7.3.2 Usability Studies

In addition, we plan to evaluate our framework in industry settings on commercial projects and use these results to further refine the framework.

7.3.3 Prediction Model Accuracy

To improve the accuracy of our bayesian network, we plan to conduct more empirical studies using different systems including commercial systems such that we have a larger dataset to build our prediction model.

REFERENCES

[1] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," *Software maintenance and reengineering (CSMR), 2011 15th European conference on*. IEEE, 2011, pp. 181–190.

[2] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an Ontology of Terms on Technical Debt," *Proceedings of the 2014 Sixth International Workshop on Managing Technical Debt*, Washington, DC, USA, 2014, MTD '14, pp. 1–7, IEEE Computer Society.

[3] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, "A bayesian belief network for assessing the likelihood of fault content," *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 2003, pp. 215–226.

[4] N. Ayewah and W. Pugh, "The google findbugs fixit," *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 241–252.

[5] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, Oct 1996, pp. 751–761.

[6] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis, "Investigating Quality Factors in Object-oriented Designs: An Industrial Case Study," *Proceedings of the 21st International Conference on Software Engineering*, New York, NY, USA, 1999, ICSE '99, pp. 345–354, ACM.

[7] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing Technical Debt in Software-reliant Systems," *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, New York, NY, USA, 2010, FoSER '10, pp. 47–52, ACM.

[8] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st edition, John Wiley & Sons, Inc., New York, NY, USA, 1998.

[9] Š. Cais and P. Pícha, "Identifying Software Metrics Thresholds for Safety Critical System," *SDIWC*, 2014.

[10] M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, Aug. 2000, pp. 786–796.

[11] C. Catal, B. Diri, and B. Ozumut, "An Artificial Immune System Approach for Fault Prediction in Object-Oriented Software," *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on*, June 2007, pp. 238–245.

[12] E. Chandra and P. E. Linda, "Class break point determination using CK metrics thresholds," *Global journal of computer science and technology*, vol. 10, no. 14, 2010.

[13] E. Charniak, "Bayesian networks without tears.," *AI magazine*, vol. 12, no. 4, 1991, p. 50.

[14] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, Jun 1994, pp. 476–493.

[15] S. Chin, E. Huddleston, W. Bodwell, and I. Gat, "The economics of technical debt," *Cutter IT Journal*, vol. 23, no. 10, 2010, pp. 11–15.

[16] Z. Codabux and B. Williams, "Agile Adoption - An Industrial Case Study," *Technical Report*, 2012, number 120731.

[17] Z. Codabux and B. Williams, "Managing Technical Debt: An Industrial Case Study," *Proceedings of the 4th International Workshop on Managing Technical Debt*, Piscataway, NJ, USA, 2013, MTD '13, pp. 8–15, IEEE Press.

[18] Z. Codabux, B. Williams, and N. Niu, "A Quality Assurance Approach to Technical Debt," *Proceedings of the International Conference on Software Engineering Research and Practice*, 2014, SERP '14, pp. 172–178.

[19] Z. Codabux, B. J. Williams, G. L. Bradshaw, and C. Murray, "An Empirical Assessment of Technical Debt Practices in Industry," *Journal of software: Evolution and Process*, 2016 (Under Review).

[20] W. Cunningham, "The WyCash Portfolio Management System," *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, New York, NY, USA, 1992, OOPSLA '92, pp. 29–30, ACM.

[21] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *IEEE Software*, vol. 29, no. 6, Nov 2012, pp. 34–42.

[22] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of Technical Debt," *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 49–53.

[23] M. R. Dale and C. Izurieta, "Impacts of design pattern decay on system quality," *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 37.

[24] A. Darwiche, *Modeling and reasoning with Bayesian networks*, Cambridge University Press, 2009.

[25] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, 2004, pp. 129–143.

[26] A. K. Deo, *Assessing software defects using nano-patterns detection*, master's thesis, Mississippi State University, 2015.

[27] N. Fenton and M. Neil, *Risk assessment and decision analysis with Bayesian networks*, CRC Press, 2012.

[28] N. Fenton, M. Neil, W. Marsh, P. Hearty, Ł. Radliński, and P. Krause, "On the effectiveness of early life cycle defect prediction with Bayesian Nets," *Empirical Software Engineering*, vol. 13, no. 5, 2008, pp. 499–537.

[29] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, 2012, pp. 244–257.

[30] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Object Technology Series. Pearson Education, 2012.

[31] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Object Technology Series. Pearson Education, 2012.

[32] B. G. Glaser and A. L. Strauss, *The discovery of grounded theory: Strategies for qualitative research*, Transaction publishers, 2009.

[33] M. W. Godfrey and E. H. S. Lee, "Secrets from the Monster: Extracting Mozilla's Software Architecture," *Proceedings of International Symposium on Constructing software engineering tools*, 2000, pp. 15–23.

[34] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software engineering*, vol. 31, no. 10, 2005, pp. 897–910.

[35] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*, Elsevier Science Inc., New York, NY, USA, 1977.

[36] D. Heckerman, D. Geiger, and D. M. Chickering, "Learning Bayesian networks: The combination of knowledge and statistical data," *Machine learning*, vol. 20, no. 3, 1995, pp. 197–243.

[37] T. T. Ho and G. Ruhe, "When-to-Release Decisions in Consideration of Technical Debt," *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, Sept 2014, pp. 31–34.

[38] L. Hochstein and M. Lindvall, "Diagnosing architectural degeneration," *Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard*, Dec 2003, pp. 137–142.

[39] J. Holvitie, V. Leppänen, and S. Hyrynsalmi, "Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey," *Proceedings of the 2014 Sixth International Workshop on Managing Technical Debt*, Washington, DC, USA, 2014, MTD '14, pp. 35–42, IEEE Computer Society.

[40] C. Izurieta and J. M. Bieman, "How Software Designs Decay: A Pilot Study of Pattern Evolution," *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Sept 2007, pp. 449–451.

[41] C. Izurieta and J. M. Bieman, "Testing Consequences of Grime Buildup in Object Oriented Design Patterns," *2008 1st International Conference on Software Testing, Verification, and Validation*, April 2008, pp. 171–179.

[42] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Software Quality Journal*, vol. 21, no. 2, 2013, pp. 289–323.

[43] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Mining the relationship between anti-patterns dependencies and fault-proneness.," *WCRE*, 2013, pp. 351–360.

[44] R. Jabangwe, J. Börstler, D. Smite, and C. Wohlin, "Empirical Evidence on the Link Between Object-oriented Measures and External Quality Attributes: A Systematic Literature Review," *Empirical Softw. Engg.*, vol. 20, no. 3, June 2015, pp. 640–693.

[45] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do Code Clones Matter?," *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, ICSE '09, pp. 485–495, IEEE Computer Society.

[46] C. J. Kapser and M. W. Godfrey, "âĂIJCloning considered harmfulâĂİ considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, 2008, pp. 645–692.

[47] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Washington, DC, USA, 2009, WCRE '09, pp. 75–84, IEEE Computer Society.

[48] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, 2012, pp. 243–275.

[49] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, no. 2004, 2004, pp. 1–26.

[50] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and software technology*, vol. 51, no. 1, 2009, pp. 7–15.

[51] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An Enterprise Perspective on Technical Debt," *Proceedings of the 2Nd Workshop on Managing Technical Debt*, New York, NY, USA, 2011, MTD '11, pp. 35–38, ACM.

[52] J.-L. Letouzey, "The SQALE method for evaluating technical debt," *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 31–36.

[53] W. Li and R. Shatnawi, "An Empirical Study of the Bad Smells and Class Error Probability in the Post-release Object-oriented System Evolution," *J. Syst. Softw.*, vol. 80, no. 7, July 2007, pp. 1120–1128.

[54] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, "An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt," *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, New York, NY, USA, 2014, QoSA '14, pp. 119–128, ACM.

[55] E. Lim, N. Taksande, and C. Seaman, "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Software*, vol. 29, no. 6, Nov 2012, pp. 22–27.

[56] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, Sept 2008, pp. 227–236.

[57] M. A. A. Mamun, C. Berger, and J. Hansson, "Explicating, Understanding, and Managing Technical Debt from Self-Driving Miniature Car Projects," *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, Sept 2014, pp. 11–18.

[58] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, Sept 2012, pp. 9:1–9:13.

[59] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, Dec 1976, pp. 308–320.

[60] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, Jan 2010, pp. 20–36.

[61] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, 2002, pp. 87–94.

[62] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "Balancing Agility and Formalism in Software Engineering," Springer-Verlag, Berlin, Heidelberg, 2008, chapter A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266.

[63] R. E. Neapolitan et al., "Learning bayesian networks," 2004.

[64] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In Search of a Metric for Managing Architectural Technical Debt," *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, Washington, DC, USA, 2012, WICSA-ECSA '12, pp. 91–100, IEEE Computer Society.

[65] A. Nugroho, J. Visser, and T. Kuipers, "An Empirical Model of Technical Debt and Interest," *Proceedings of the 2Nd Workshop on Managing Technical Debt*, New York, NY, USA, 2011, MTD '11, pp. 1–8, ACM.

[66] A. Okutan and O. T. Yıldız, "Software defect prediction using Bayesian networks," *Empirical Software Engineering*, vol. 19, no. 1, 2014, pp. 154–181.

[67] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems," *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, Washington, DC, USA, 2009, ESEM '09, pp. 390–400, IEEE Computer Society.

[68] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are All Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems," *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, Washington, DC, USA, 2010, ICSM '10, pp. 1–10, IEEE Computer Society.

[69] R. L. Ott and M. T. Longnecker, *An introduction to statistical methods and data analysis*, Cengage Learning, 2008.

[70] G. J. Pai and J. B. Dugan, "Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, Oct 2007, pp. 675–686.

[71] E. Pérez-Miñana and J.-J. Gras, "Improving fault prediction using Bayesian networks for the development of embedded software applications," *Software testing, verification and reliability*, vol. 16, no. 3, 2006, pp. 157–174.

[72] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes," *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 437–446.

[73] T. L. Saaty, "The Analytic Hierarchy Process, New York: McGrew Hill," *International, Translated to Russian, Portuguesses and Chinese, Revised edition, Paperback (1996, 2000), Pittsburgh: RWS Publications*, 1980.

[74] K. Schmid, "A Formal Approach to Technical Debt Decision Making," *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, New York, NY, USA, 2013, QoSA '13, pp. 153–162, ACM.

[75] K. Schwaber and J. Sutherland, *Software in 30 Days: How Agile Managers Beat the Odds, Delight Their Customers, And Leave Competitors In the Dust*, Soft-ware in 30 Days: How Agile Managers Beat the Odds, Delight Their Customers, and Leave Competitors in the Dust. Wiley, 2012.

[76] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetrò, "Using Technical Debt Data in Decision Making: Potential Decision Approaches," *Proceedings of the Third International Workshop on Managing Technical Debt*, Piscataway, NJ, USA, 2012, MTD '12, pp. 45–48, IEEE Press.

[77] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned About Fighting Defects," *Proceedings of the 8th International Symposium on Software Metrics*, Washington, DC, USA, 2002, METRICS '02, pp. 249–, IEEE Computer Society.

[78] V. Singh, W. Snipes, and N. A. Kraft, "A Framework for Estimating Interest on Technical Debt by Monitoring Developer Activity Related to Code Comprehension," *Proceedings of the 2014 Sixth International Workshop on Managing Technical Debt*, Washington, DC, USA, 2014, MTD '14, pp. 27–30, IEEE Computer Society.

[79] Y. Singh, A. Kaur, and R. Malhotra, "Empirical validation of object-oriented metrics for predicting fault proneness models," *Software quality journal*, vol. 18, no. 1, 2010, pp. 3–35.

[80] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the Effect of Code Smells on Maintenance Effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, Aug 2013, pp. 1144–1156.

[81] W. Snipes, B. Robinson, Y. Guo, and C. Seaman, "Defining the Decision Factors for Managing Defects: A Technical Debt Perspective," *Proceedings of the Third International Workshop on Managing Technical Debt*, Piscataway, NJ, USA, 2012, MTD '12, pp. 54–60, IEEE Press.

[82] R. O. Spínola, N. Zazworka, A. Vetrò, C. Seaman, and F. Shull, "Investigating technical debt folklore: Shedding some light on technical debt opinion," *Managing Technical Debt (MTD), 2013 4th International Workshop on*, May 2013, pp. 1–7.

[83] J. V. Stone, *Bayes' rule: a tutorial introduction to Bayesian analysis*, Sebtel Press, 2013.

[84] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting Bugs Using Antipatterns.," *ICSM*, 2013, vol. 13, pp. 270–279.

[85] R. T. Tvedt, P. Costa, and M. Lindvall, "Does the code match the design? A process for architecture evaluation," *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 393–401.

[86] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, 1995, p. 11.

[87] L. Williams and A. Cockburn, "Agile software development: it's about feedback and change," *Computer*, vol. 36, no. 6, June 2003, pp. 39–43.

[88] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 411–420.

[89] J. Yli-Huumo, A. Maglyas, and K. Smolander, "Product-Focused Software Process Improvement: 15th International Conference, PROFES 2014, Helsinki, Finland, December 10-12, 2014. Proceedings," 2014, pp. 93–107.

[90] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, et al., "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, 2014, pp. 403–426.

[91] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the Impact of Design Debt on Software Quality," *Proceedings of the 2nd Workshop on Managing Technical Debt*, New York, NY, USA, 2011, MTD '11, pp. 17–23, ACM.

[92] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Prioritising Refactoring Using Code Bad Smells," *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 458–464.

[93] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: A Review of Current Knowledge," *J. Softw. Maint. Evol.*, vol. 23, no. 3, Apr. 2011, pp. 179–202.

[94] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*. IEEE, 2007, pp. 9–9.

APPENDIX A

INDUSTRIAL CASE STUDY

Industrial Case Study - Interview Questions [17]

Section 1: Technical debt

1. How would you define/describe technical debt?

2. How important is lowering technical debt in the organization? Why? Do you think your team does a good job addressing technical debt?

Section 2: Categories

1. Debts at the organization are typically classified as automation debt and infrastructure debt. How would you classify the debts that you address?

Section 3: Cost Estimation

1. Does your team incur debt intentionally? If yes, why? What are the benefits of doing so?

2. Do you record how much time you spend reducing debt?

3. If yes, how much time do you spend reducing debt?

Section 4: Prioritize/decision making

1. What type of debt is most difficult to address?

2. What methodology do you use to track debt?

3. How do you prioritize technical debt?

4. What are the impacts of technical debt for (1) the team (2) the customers (3) future modifications of the system?

5. How much time can a debt be on hold in the backlog?

Section 5: Demographic Questions

1. What system does your team work on?

2. What is your role in on the team?

3. What are your responsibilities?

4. How many people are on your team?

5. How many years of work experience do you have? (in and outside of the company)?

6. How many years of Agile work experience do you have?

7. What academic degree do you have?

8. Are team members geographically distributed?

9. How often do members of the development team interact with stakeholders?

10. Did you receive any formal Agile training? Duration of training?

APPENDIX B

SURVEY

Survey [19]

B.1 Interviews (Pre-Survey)

Section 1: Definition

1. How do you define technical debt (TD)?

2. Do you distinguish between quality debt (i.e., defects) and technical debt? Do you consider both types of debt?

3. What are your thoughts on the statement: "Cost associated with TD goes beyond effort to address the debt. There is a need to include cost associated with company liability, reputation, market share, etc." (e.g., TD as the liability incurred for shipping software)?

Section 2: Quantification

1. Do you measure TD? If so, what techniques and metrics are used in measuring TD?

2. How do you estimate the value (benefit, not cost) of incurring TD?

Section 3: Intentionality

1. Do you incur TD intentionally? If so, in what circumstances?

2. What are the benefits of doing so?

3. What are the negative impacts of incurring intentional TD?

4. How do you account for risk when incurring intentional TD?

5. Have you found that you incur debt unintentionally?

Section 4: Assessment

1. How do you know when TD is a problem?

Section 5: Communication

1. Do you communicate TD to stakeholders (customer, team members, management)? How?

B.2 Hypotheses

The following hypotheses were generated from the interview results.

*H*1 *Practitioners agree about what constitutes technical debt*

*H*2 *There are established units of technical debt*

*H*3 *There are agreed methods to reduce technical debt*

*H*4 *Practitioners agree that technical debt is distinct from defects*

*H*5 *Most practitioners consider the additional costs involved in reducing technical debt other than effort*

*H*6 *Practitioners discuss the benefits of intentionally incurring technical debt*

*H*7 *Practitioners assess the risks associated with intentional technical debt*

B.3 Survey

Part I - Technical Debt

1. Which statement do you most agree with concerning technical debt? (First Choice)

(A) Technical debt is a tradeoff made due to schedule, scope, cost, quality constraints

(B) Technical debt is the additional cost to maintain and update the software to ensure a high level of quality and a reduced number of defects

(C) Technical debt is a shortcut taken during software development that makes subsequent changes to the software harder

(D) Technical debt is a code deficiency that preserves the functionality of the software but hinder its maintainability

(E) Other:

2. Which statement do you most agree with concerning technical debt? (Second Choice)

(A) Technical debt is a tradeoff made due to schedule, scope, cost, quality constraints

(B) Technical debt is the additional cost to maintain and update the software to ensure a high level of quality and a reduced number of defects

(C) Technical debt is a shortcut taken during software development that makes subsequent changes to the software harder

(D) Technical debt is a code deficiency that preserves the functionality of the software but hinder its maintainability

(E) Other:

3. Rate the difficulty of addressing each of the following types of technical debt (a-d):

(a) Architecture/Design Debt (when the software design no longer fits its intended purpose)

Least Difficult ... Most Difficult

1 2 3 4 5

(b) Code Debt (violations of design principles in production code)

Least Difficult ... Most Difficult

1 2 3 4 5

(c) Test Debt (test plan is not completely carried out)

Least Difficult ... Most Difficult

1 2 3 4 5

(d) Test Debt (test plan is not completely carried out)

Least Difficult ... Most Difficult

1 2 3 4 5

4. Please rate your agreement to the following statement:

   The technical debt landscape should include requirements, documentation and infrastructure debts.

   (Requirements debt refers to tradeoffs made with respect to what requirements the development team need to implement. Documentation debt is missing or inadequate documentation. Infrastructure debt is delaying an upgrade or infrastructure fix).

   O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

5. Do you consider process debts to be one type of technical debt or totally distinct from technical Debt? (Process debt refers to inefficient processes, e.g. what the process was designed to handle may be no longer appropriate)

O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

6. Do you consider defects to be one type of technical debt or totally distinct from technical Debt?

   One Type ... Totally Distinct

   1 2 3 4 5

7. My department quantifies the amount of technical debt it owns

   O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

8. My department measures technical debt using the following criteria

   ☐ Story points

   ☐ Person hours required to pay the debt

   ☐ Cost in dollars or other currency

   ☐ My department does not measure its technical debt

   ☐ Other:

9. My department has a systematic means of reducing technical debt

   O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

10. My department measures technical debt using the following criteria

☐ Regularly refactors its source code

☐ Allocates time in each development cycle to address technical debt

☐ Employs teams primarily focused on testing and test debt

☐ Employs teams primarily focused on fixing defects

☐ Hires / consults external development teams to address technical debt issues

☐ Relegates technical debt as an individual developer responsibility

☐ Other:

11. My department quantifies the amount of technical debt it owns

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

12. My department have a central mechanism to track technical debt

    ☐ Issue tracking system

    ☐ Developer TODO lists

    ☐ Sticky notes / developer note pads

    ☐ Other:

13. My department regularly prioritizes its technical debt using some defined criteria

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

14. My department uses the following criteria to prioritize its technical debt

☐ Cost to pay off

☐ Urgency of customer request

☐ Amount of risk involved

☐ Potential to significantly impact customer base

☐ We do not have any established criteria for prioritizing technical debt

☐ Other:

15. Most companies are unaware of the amount of technical debt owned by a company it is aquiring

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

16. Most companies are unaware of the amount of technical debt contained in its legacy systems

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

17. Technical debt is viewed differently in startup / early-stage companies compared to well-established larger companies

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

18. Technical debt is viewed differently in shrink-wrapped software versus software as-a-service offerings.

(Shrink-wrapped software are boxed versions of computer software purchased at a store. Software-as-a-service (or SaaS) is a way of delivering applications by a vendor or service provider over a network (typically the Internet) to customers.)

O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

19. The cost of technical debt is primarily the cost associated with the effort to reduce the debt

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

20. Technical debt cost considerations should factor in more than just the cost of effort to pay the debt in person-hours

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

21. Please describe any other considerations when assessing the cost of technical debt in a system

    _____

    _____

22. My development team members discuss the benefits of intentionally incurring technical debt

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

23. Most of the development teams in my company discuss the tradeoffs of intentionally incurring technical debt

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

24. My development team members assess the risks of intentionally incurring technical debt

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

25. Most of the development teams in my company assess the risks of intentionally incurring technical debt

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

26. Evaluating technical debt should be part of risk assessment in a company

    O Strongly disagree O Disagree O Neither agree nor disagree O Agree O Strongly agree

27. Please provide any additional comments, concerns, or issues regarding technical debt

    _____

    _____

Part II - Demographic Information

1. What is the name of the company that you work for?

   (Company information will not be reported in the study results. This information is obtained to account for potential bias in the responses)

   _____

2. Country of Location

   _____

3. Number of Employees (Estimates are fine)

   (A) 1 - 99

   (B) 100 - 999

   (C) 1000 - 4999

   (D) > 5000

   (E) Don't know

4. Size of your department

   (A) 1 - 9

   (B) 10 - 49

   (C) 50 - 99

   (D) 100 - 499

   (E) > 500

   (F) Don't know

5. How many years of experience do you have in software development?

   (A) 0 - 2

   (B) 3 - 5

   (C) 6 - 10

   (D) 11 - 20

   (E) > 20

6. What is your role in the department?

   ☐ Software Developer

   ☐ Software Architect (Design)

   ☐ Requirements Analyst

   ☐ Quality Analyst / Testing

   ☐ Project Manager

   ☐ User Interface Development

   ☐ Other:

7. How would you categorize the types of systems being developed by your team?

   _____

   _____

8. Age of the primary system under your responsibility

   (A) 0 - 2

   (B) 3 - 5

(C) 6 - 10

(D) > 10